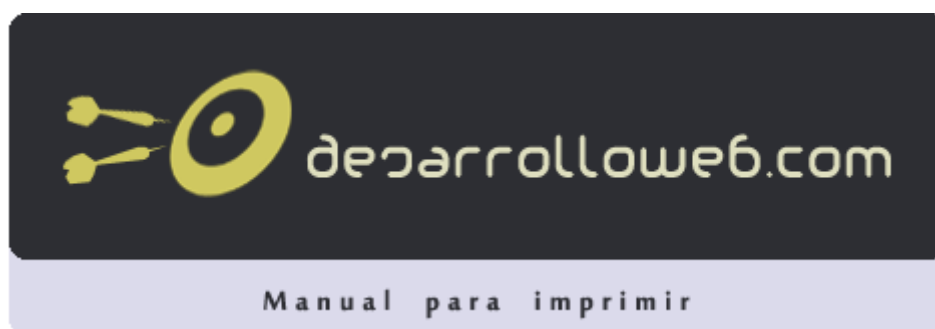


# Manual de Laravel 5

*Manual del framework PHP Laravel, centrándonos en versión Laravel 5 (concretamente Laravel 5.1), que nos trae diversas mejoras en rendimiento y cambios en la organización de los archivos y proyectos. Es un manual que explica paso por paso los elementos más importantes que forman parte de este popular framework, a la vez que nos ofrece una guía para comenzar a crear aplicaciones web basadas en él.*

*Comenzamos con la instalación, usando una máquina virtual llamada Homestead, que es la plataforma oficial de desarrollo de Laravel 5. Luego usamos Composer para bajarnos e instalar el framework y a partir de ahí ya nos dedicamos a analizar el framework en detalle.*

*En este manual usamos la versión de Laravel 5.1 que además es la primera versión del framework ofrecida como LTS (Long Term Support), lo que te asegura actualizaciones de seguridad para los próximos años.*



## Autores del manual

Este manual ha sido realizado por los siguientes colaboradores de DesarrolloWeb.com:

### **Carlos Ruiz Ruso**

Consultor tecnológico para el desarrollo de proyectos online especializado en WordPress y el framework Laravel.

<http://micromante.com/>

(7 capítulos)

### **Miguel Angel Alvarez**

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.

<http://www.desarrolloweb.com>

(8 capítulos)

# Parte 1:

# Instalación y configuración de Laravel 5

En esta primera parte del manual te enseñamos a instalar Laravel 5 y a configurar tu entorno de trabajo. Abordamos con detalle la plataforma oficial para desarrollo de Laravel, basada en una máquina virtual Vagrant, con el nombre de Homestead. Esta alternativa te ofrece las condiciones para desarrollar más parecidas a un entorno de producción y es la opción recomendada para tu entorno de desarrollo, que te ayudará a familiarizarte con la administración de servidores y evitará problemas cuando publiques tu web en el servidor definitivo.

## 1.1.- Homestead de Laravel

*Procedimiento de instalación del entorno de desarrollo ideal para la creación de aplicaciones usando el framework PHP Laravel y la virtualización de una máquina con Vagrant y Homestead.*

En este artículo vamos a ofrecer las pautas para contar con el mejor entorno de desarrollo posible para Laravel 5, el más profesional, fácil de instalar y de mayor versatilidad, ayudándonos de la virtualización de la máquina de desarrollo y Vagrant.

La fórmula tiene el nombre de Homestead, una herramienta oficial creada por el equipo de Laravel para facilitar la vida a los desarrolladores que quieran usar este framework PHP para comenzar un proyecto rápidamente.

La idea es que los desarrolladores tengamos que implementar el menor número de configuraciones en nuestras máquinas y que con dos o tres comandos podamos conseguir todo lo que necesitaría el entorno de desarrollo más exigente.

**Nota:** Con este texto damos inicio al [Manual de Laravel 5](#), en el que pretendemos enseñar a los lectores a trabajar con este importante

framework PHP. Así que siéntate cómodo y aprovecha para comenzar con nosotros esta zambullida en profundidad a Laravel 5.1.



### 1.1.1.- Qué es exactamente Homestead

Es una "Box de Vagrant". Por si no lo sabes, [Vagrant es una capa por encima de Virtualbox](#) (o de otros sistemas de virtualización como VMWare) que nos sirve para crear entornos de desarrollo y las Box en su terminología son imágenes de sistemas operativos ya instalados. Generalmente vienen "peladas", tal como tendríamos el ordenador si hubiéramos acabado de instalar el sistema operativo en ese momento. O en algunos casos, como Laravel Homestead, ya configurados con distintos paquetes de software.

Homestead te instala software como: Ubuntu, PHP, HHVM, Nginx, MySQL, PostgreSQL, NodeJS, Redis y algunas librerías como Memcached, Laravel Envoy, Beanstalkd...

### 1.1.2.- Por qué Homestead

Porque la comunidad de creadores de Laravel quieren acercarte al framework. Facilitarte no solo la entrada como desarrollador de Laravel, sino también proporcionarte el entorno de desarrollo más adecuado para acometer cualquier tipo de proyecto.

Realmente puedes contar con Laravel funcionando en un Xampp, Mamp o similares, pero si virtualizas tendrás varias ventajas como contar con un entorno mucho más parecido al de producción, independiente de otros proyectos con los que estés desarrollando y además el mismo sistema, con los mismos paquetes de software. No tiene sentido extenderse mucho en las [ventajas de los entornos virtualizados, porque ya hemos hablado de ellos en otras ocasiones](#). Pero piensa además que puedes conseguir todo eso con la facilidad de lanzar unos pocos comandos para generar automáticamente toda la infraestructura perfectamente configurada.

### 1.1.3.- Qué necesitas

Instalar VirtualBox como sistema de virtualización, o VMware si lo prefieres (aunque en ese caso tendrías que instalar un plugin adicional). Luego instalas Vagrant. Ambos programas son gratuitos y multiplataforma.

Una vez instalados ambos programas (Primero se recomienda instalar Virtualbox y luego Vagrant) se debe descargar la caja "Box" de Homestead, para lo que lanzarás el siguiente comando de consola.

```
vagrant box add laravel/homestead
```

Ese comando lo podrás lanzar desde cualquier carpeta, desde el terminal de línea de comandos, claro está.

**Nota:** Si tienes una versión antigua de Vagrant quizás tengas que darle la URL completa de la localización de la Box.

```
vagrant box add laravel/homestead https://atlas.hashicorp.com/laravel/boxes/homestead
```

### 1.1.4.- Instalar Homestead

El tercer paso sería instalar el propio Homestead. Para ello usaremos la box que acabamos de descargar y Vagrant. Pero necesitamos un archivo de configuración que se encuentra en Github que contiene toda la información de configuración que Vagrant necesitaría para generar y aprovisionar la máquina virtual para desarrollar con Laravel.

Para ello abrimos un terminal y nos situamos en nuestra carpeta de proyectos habitual y desde allí clonaremos el repositorio de Homestead con Git.

```
git clone https://github.com/laravel/homestead.git Homestead
```

**Nota:** Ese comando clona un repositorio. El repositorio está en Github en la ruta <https://github.com/laravel/homestead.git>. El

parámetro final del comando anterior `git clone` que está definido como "Homestead" es la carpeta donde va a colocar los archivos del

repositorio que estás clonando. Si quisieras podrías cambiarle el nombre a la carpeta de destino, por ejemplo, "proyecto-prueba" en vez

de "Homestead", para lo que el comando sería el siguiente:

```
git clone https://github.com/laravel/homestead.git proyecto-prueba
```

La estructura de carpetas que tengas tú en tu ordenador de momento es indiferente y responderá a tus propias preferencias. Lo que te

interesa es todo lo que tiene el repositorio, que te vas a descargar con el comando "git clone". El lugar donde esté localizada la carpeta

donde te descargas el repo de Homestead es lo de menos, por ejemplo muchas personas prefieren colocarla en la carpeta "Dropbox" para

que cuando se realicen cambios en el proyecto se realicen copias de seguridad automáticas en la nube.

Dentro de ese repositorio clonado tenemos en un archivo bash con el nombre "init.sh" que tiene el código con los comandos para crear el archivo de configuración Homestead.yaml. Simplemente lo tienes que ejecutar, con el comando:

```
bash init.sh
```

Obviamente el comando "bash init.sh" lo debes ejecutar desde la ruta donde se encuentra el archivo init.sh, en la carpeta del repositorio que acabas de clonar. Verás que simplemente te da un mensaje de respuesta "Homestead initialized!". Eso quiere decir que ha ido bien y ha podido crear los archivos de configuración necesarios para los siguientes pasos.

Entre los archivos de configuración que se crean al ejecutar el init.sh hay uno que necesitarás editar, llamado "**Homestead.yaml**". Este archivo de configuración lo encontrarás en tu carpeta personal (la de tu usuario dentro de tu sistema operativo) y dentro de ella en la carpeta oculta ".homestead". Si lo abres podrás encontrar los datos de configuración de la máquina virtual Homestead, el servidor de desarrollo virtualizado que vamos a crear, que incluye datos como la IP del servidor, las CPUs, memoria asignada, carpetas que se compartirán entre la máquina de desarrollo

y el ordenador anfitrión (nuestro ordenador físico) y cosas así.

**Nota:** Por ejemplo en Mac mi directorio del usuario personal es /Users/midesweb y la carpeta .homestead estará por tanto en

/Users/midesweb/.homestead

En un principio necesitamos tocar pocas cosas del Homestead.yaml, porque la mayoría ya se ha configurado automáticamente, acorde con nuestro sistema operativo gracias al archivo init.sh, pero podrás desear cambiar los siguientes item:

- IP del servidor virtual, por defecto aparece ip: "192.168.10.10".
- Llave para la conexión SSH con la máquina virtual (ver información más abajo)
- Folders, que son las carpetas que vas a tener en tu sitio, mapeadas en dos direcciones, el servidor virtual y el ordenador local (luego te decimos cómo).
- Sites, para definir el nombre del sitio para el virtualhost.

### 1.1.5.- Crear la llave de SSH

Esta es la llave para la conexión con el servidor virtual por línea de comandos. Si no tienes una la puedes crear. Desde Mac o Linux es tan sencillo como lanzar el siguiente comando:

```
ssh-keygen -t rsa -C "you@homestead"
```

Desde Windows la recomendación es usar el "Bash de Git" (el programa de terminal que te viene incorporado cuando instalas Git) y lanzar el mismo comando anterior. Aunque también podrías usar el popular software PuTTY y PuTTYgen.

En el anterior comando no necesitas editar nada, lo pones tal cual. Enseguida verás que el programa que te genera la clave te pide la ruta donde va a colocar esta key generada. El propio programa te muestra una ruta predeterminada donde sugiere se coloquen las llaves. Podemos pulsar enter sin escribir nada para aceptar esa ruta o bien escribir cualquier otra ruta donde se generará la clave. Puedes poner lo que quieras, con tal que te acuerdes de la ruta que has seleccionado.

**Nota:** en cuanto a la ruta donde se van a colocar las claves es totalmente personalizable. Pon la que quieras, simplemente acuérdate

donde están los archivos. Para ser más específicos y por si alguien se pierde, pueden darse tres casos:

- Que te parezca bien la ruta donde te sugiere que va a crear las claves. En ese caso puedes apretar enter y listo. En mi caso la

ruta donde se colocarían las claves de manera predeterminada es /Users/midesweb/.ssh/id\_rsa. Por tanto se crearán los

archivos id\_rsa e id\_rsa.pub en la carpeta /Users/midesweb/.ssh.

- Que escribas el nombre de un archivo simplemente, algo como "homesteadkey". En ese caso te creará las claves en el directorio

donde estás situado cuando hiciste el comando ssh-keygen, archivos homesteadkey y homesteadkey.pub.

•Que que escribas una ruta absoluta, algo como /Users/midesweb/.ssh/nuevaclave, en cuyo caso te colocará la clave en esa ruta

absoluta, tanto la clave privada /Users/midesweb/.ssh/nuevaclave como la pública /Users/midesweb/.ssh/nuevaclave.pub

El programa de generar las key te pedirá también insertar una frase como clave, que puedes poner la que desees. O bien una cadena vacía si quieres ahorrarte indicar una frase que te puedas olvidar. Como es solo para trabajar en local sería suficiente.

Como decíamos, una vez generada la clave lo que te interesa es saber la ruta donde se ha colocado, que tendrás que usar para configurar el archivo Homestead.yaml.

El lugar donde vas a tener que introducir la ruta de la clave será algo como esto:

```
authorize: ~/.ssh/id_rsa.pub
```

```
keys:  
- ~/.ssh/id_rsa
```

Esa ruta es la que tienes que configurar según la localización de las llaves creadas. Generalmente si seleccionaste la carpeta predeterminada que sugería el comando ssh-keygen ni siquiera tendrías que editar nada. Si las claves las guardaste en otro lugar, entonces tendrás que indicarlo aquí. De todos modos, no tiene ninguna complicación, es saber la ruta donde está tu clave e indicarla en el Homestead.yaml.

**Nota:** Recuerda que ~/ es un alias de tu ruta personal (ver otra nota más abajo donde se habla sobre este tema). Pero si te lía lo puedes

cambiar por una ruta absoluta y punto.

### 1.1.6.- Carpetas del servidor virtual y mapeo a las carpetas locales

Una de las características de los entornos con Vagrant es que hay determinadas carpetas del servidor virtual (invitado) que están mapeadas a directorios locales de tu máquina real (anfitrión). Esto te permite editar archivos de tu proyecto como los editarías en cualquier otra situación, con tus editores de código preferidos, trabajando en local como si lo hicieras con un sistema más tradicional tipo Xampp o Mamp, Wamp, etc. Las carpetas locales están enlazadas con las carpetas del servidor, por lo que no tienes que subir los archivos por FTP, SCP ni nada parecido. Cuando modifiques algún archivo, o crees archivos nuevos en tu carpeta local, automáticamente esos cambios serán producidos en las carpetas enlazadas del servidor virtualizado. Podrías configurar tantas carpetas compartidas como necesites en tu proyecto.

Ahora en el archivo Homestead.yaml tendrás que definir esa estructura de carpetas, que se adaptará a tus costumbres habituales. Como hemos comentado antes, tendrás una carpeta de tu ordenador donde guardas tus proyectos y seguirá siendo así. Por tanto, es simplemente guardar este proyecto en una carpeta en una ruta habitual para ti.

```
folders:  
- map: ~/carpeta_de_proyectos/proyecto_x/codigo  
  to: /home/projects
```

En este lugar colocamos en "map" el valor de nuestra carpeta en el ordenador anfitrión y en "to" la ruta de la carpeta del ordenador virtualizado que vamos a crear con Vagrant y Homestead. La carpeta que estás indicando en "map", que corresponde con tu ordenador real, debería existir.

**Nota:** Una aclaración acerca de las rutas para los de Windows. "~/ " corresponde con la carpeta raíz de tu usuario. Por ejemplo, si tu

usuario (con el que entras en Windows es "micromante" esa ruta correspondería con una carpeta como "C:/Users/micromante". La

carpeta "~/homestead" correspondería físicamente con "C:/Users/micromante/homestead". Si usas el terminal básico del Windows no

reconocerá la ruta ~/, así que tendrás que usar la ruta real de tu carpeta de usuario, la física o ruta real. Si usas un terminal un poco mejor,

por ejemplo el Git Bash sí que te reconocerá ese nombre lógico ~/ y te llevará a tu carpeta de usuario, sea cual sea tu nombre de usuario,

igual que ocurre en el terminal de Linux o Mac.

De nuevo, no hay necesidad de colocar algo específico en la sección folders, sino que será tal como tú desees organizar las carpetas, tanto en el servidor Homestead como en tu ordenador local.

### 1.1.7.- Definir la configuración de los sitios que vamos a albergar en esta máquina Homestead

Nos queda un valor que configurar, que es el nombre del host virtual que nos va a configurar Homestead en esta máquina virtual que estamos a punto de crear. Este host virtual o "virtualhost" lo usaremos para entrar en el proyecto desde un navegador, con un nombre de dominio, como si fuera un servidor remoto. Lógicamente ese dominio solo estará disponible desde tu ordenador y tendrás que configurar el conocido archivo "hosts" para asociar la IP del servidor al nombre de dominio virtual.

sites:

```
- map: homestead.dw  
  to: /home/projects/mi_proyecto/public
```

En este caso en "map" colocamos el nombre de dominio para el virtualhost y en "to" indicamos la carpeta del servidor virtual donde va a estar el "document root". Si te fijas, ese document root debe colgar de la carpeta que tienes en la máquina virtual que has enlazado con tu carpeta real, de ese modo podrás editar en local los archivos de tu proyecto que se servirá desde el ordenador virtual.

**Nota:** En una instalación de Homestead serías capaz de albergar varios proyectos si así lo deseas, con diversos host virtuales.

Insistimos en que cada uno de los host virtuales tendrás que configurarlo en el archivo de hosts de tu ordenador, asociando la IP que

hayas configurado para esta máquina virtual con el nombre de dominio de tus virtualhost.

Tu archivo hosts tendrás que editarlo agregando esta líneas, que quedará más o menos de esta forma:

```
192.168.10.10    homestead.dw
```

Aunque lógicamente, tendrás que poner la IP que hayas definido en el campo IP de Homestead.yaml y el nombre de dominio que hayas definido en los "sites", en el campo "map".

**Nota:** La configuración del archivo Hosts está explicada en el artículo: [Modificar el archivo de Hosts en Windows, Linux y Mac](#)

### 1.1.8.- Lanzar la máquina virtual

Una vez configurado el Homestead.yml tienes que lanzar la máquina virtual, es decir, tendrás que ponerla en marcha, con el conocido comando de Vagrant:

```
vagrant up
```

Ese comando lo tienes que lanzar desde la carpeta donde has puesto el repositorio de Homestead, que clonaste con Git. La primera vez que se ejecute va a tardar un poco, porque tiene que instanciar y configurar toda la máquina, pero luego iniciar la máquina virtual será muy rápido, también con el comando "vagrant up".

**Nota:** En los comentarios del artículo nos han preguntado en qué momento hacer el "vagrant init". Ese comando no lo necesitas hacer

tú mismo. Sirve para que se cree el Vagrantfile y configurar una carpeta para que albergue una máquina virtual que vas a crear con

Vagrant. Pero básicamente el Vagrantfile es lo que te descargas al clonar el repositorio y las configuraciones que necesita Vagrant y que

realizas predeterminadamente al hacer al "Vagrant init" son las que has aprendido a definir en el archivo Homestead.yml.

Por tanto, si en algún momento haces un "vagrant up" y te pide hacer un "vagrant init" es que estás haciendo el "vagrant up" desde una

carpeta incorrecta. En ese caso fíjate que estés situado en el repositorio de Homestead que has clonado. En esa carpeta debería haber un

Vagrantfile, ya que es uno de los archivos del repositorio clonado.

Una vez lanzada la máquina virtual podríamos hacer una primera prueba accediendo a la IP del servidor virtual con tu navegador.

```
http://192.168.10.10/
```

(Ojo, la URL podría ser otra si tocaste en el Homestead.yml el dato ip: "192.168.10.10". Si no es esa, será la IP que hayas configurado en tu caso)

De momento no veremos nada, pero al menos el navegador no deberías recibir tampoco el típico error del navegador para informar que el host no ha sido encontrado.

Ahora se trataría de colocar un archivo dentro de la carpeta que has definido como raíz de tu dominio en el campo "to:" de la configuración "sites". Recuerda que esa carpeta está enlazada desde el servidor virtual con una ruta en tu directorio de los proyectos, tal como se ha configurado.

Puedes colocar un index.php con un "Hola mundo" para probar. Lo colocarás en una ruta tal como lo hayas configurado en tu caso, siguiendo los valores definidos en este artículo nos quedaría una como esta:

Ordenador virtualizado:

```
/home/projects/mi_proyecto/public
```

Equivalente en el host anfitrión (ordenador real):

```
~/carpeta_de_proyectos/codigo/mi_proyecto/public
```



Ahora accediendo a la IP del servidor podrías encontrar tu index.php. Además, si has podido configurar correctamente tu virtualhost, podrás acceder a partir del nombre del dominio creado en tu máquina.

`http://homestead.dw/`

Con esto es todo, de momento tenemos nuestra máquina funcionando, lista para instalar Laravel, pero aún nos queda toda la [instalación del framework PHP Laravel](#) y algún trabajo más para comenzar a desarrollar. Lo veremos en próximos artículos.

Antes de acabar sería solo comentar que [tenemos un #programadorIO sobre Laravel Homestead](#) que amplía la información relatada en este artículo. Recuerda también que puedes encontrar la [documentación oficial de Homestead para Laravel 5 en este enlace](#).

### 1.1.9.- Código completo del Homestead.yaml

En los comentarios del artículo nos han pedido reproducir el código completo de un archivo Homestead.yaml. La verdad es que no lo habíamos hecho porque las configuraciones dependen mucho de las preferencias, costumbres, sistemas operativos y rutas de cada uno, por lo que no sería adecuado hacer un "copia-pegar" del archivo. Para evitar confusiones y posibles copias literales lo habíamos dejado pasar. No obstante, para que sirva de referencia para comparar vuestros archivos con los nuestros, dejo aquí el código completo.

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
  - ~/.ssh/id_rsa

folders:
  - map: ~/html/Homestead/codigo
    to: /home/projects

sites:
  - map: homestead.dw
    to: /home/projects/homestead/public
  - map: testlaravel.com
    to: /home/projects/testlaravel/public

databases:
  - homestead

variables:
  - key: APP_ENV
    value: local

# blackfire:
#   - id: foo
#     token: bar
#     client-id: foo
#     client-token: bar
```

```
# ports:
#   - send: 93000
#     to: 9300
#   - send: 7777
#     to: 777
#   protocol: udp
```

Artículo por Carlos Ruiz Ruso

## 1.2.- Instalar Laravel 5

*Tutorial para aprender a instalar el popular framework PHP Laravel 5, usando Composer.*

En este artículo del [Manual de Laravel](#) vamos a abordar la instalación del framework PHP Laravel 5 usando la conocida herramienta Composer, el gestor de dependencias de PHP, que si no conoces de antemano te recomendamos estudiar con el [Manual de Composer de DesarrolloWeb.com](#).

Además usaremos una máquina virtual para instalar Laravel, que es la denominada Homestead, la plataforma oficial de desarrollo de Laravel 5, que ya te explicamos en el [artículo sobre Homestead](#). No obstante, lo cierto es que este proceso de instalación lo podrías realizar sobre cualquier ordenador, usando virtualización o usando PHP instalado a mano o con softwares como Xampp o Mamp. El proceso es exactamente el mismo, puesto que se usa Composer también y éste es independiente de la máquina donde lo tengamos instalado, su sistema operativo, etc.



Al final del artículo además encontrarás un vídeo que te explica el proceso de manera visual y agrega otra serie de informaciones de utilidad acerca de la instalación de Laravel 5.

### 1.2.1.- Requisitos

Para instalar Laravel 5 necesitas estos requisitos:

- PHP 5.5.9
- Las extensiones de PHP:
  1. OpenSSL
  2. Mbstring
  3. Tokenizer

Además requiere que tengas instalado en tu sistema el gestor de dependencias Composer, ya que Laravel lo usa para la instalación.

Como hemos dicho anteriormente también, instalando la máquina virtual Homestead te aseguras de tener todo lo necesario para instalar Laravel 5 sin necesidad de configurar a mano ninguno de los paquetes o extensiones necesarias.

### 1.2.2.- Iniciar la máquina virtual Homestead

Esta instalación la vamos a realizar usando Homestead, así que os dejamos un par de apreciaciones que necesitarás tener en cuenta.

Debes arrancar la máquina virtual, para lo que te situarás en la línea de comandos en la carpeta donde instalaste Homestead. Desde allí lanzarás el comando:

```
vagrant up
```

Seguidamente tendrás que conectarte por SSH con la máquina virtual de Homestead, porque la instalación de Laravel la vamos a realizar sobre esa máquina y no sobre tu ordenador "real". Lo consigues con el comando:

```
vagrant ssh
```

**Nota:** Si por despiste intentas instalar Laravel 5 más adelante sin haber entrado en la máquina virtual de Homestead por SSH, lo más

seguro es que por un tema de dependencias no puedas descargarte el instalador de Laravel y te arrojará un error parecido al siguiente:

```
[RuntimeException] Could not load package guzzlehttp/guzzle in http://packagist.org: [UnexpectedValueException] Could not parse
```

```
version constraint ^1.1: Invalid version string "^1.1"
```

```
[UnexpectedValueException] Could not parse version constraint ^1.1: Invalid version string "^1.1"
```

Luego tendrás que recordar cómo instalaste Homestead y las carpetas de proyecto que fueron configuradas, así como el virtualhost que fue definido (datos de configuración "folders" y "sites"). Eso se definió en el archivo Homestead.yaml que tienes en la carpeta: ~/.homestead.

**Nota:** Fíjate que la carpeta ~/.homestead. es una carpeta oculta en Linux o Mac, por lo que para localizarla tendrás que listar archivos

ocultos con "ls -la". Los que estáis en Windows lo tenéis en esa misma carpeta, en la home de tu usuario de Windows (es lo que significa

"~/"), la diferencia es que .homestead no será una carpeta oculta.

### 1.2.3.- Instalación

Se trata de un par de sencillos pasos (sencillos ya que son comandos de Composer, que es quien hace el trabajo bruto para ti).

**1.- Primero descargamos el instalador de Laravel** y lo disponibilizamos de manera global. Este es un paso que

tendrás que hacer una vez únicamente, independientemente del número de instalaciones de Laravel que quieras crear en una máquina.

composer global require "laravel/installer=~1.1"

Recuerda que ese comando lo lanzas en la máquina virtual Homestead!!

**Nota:** Si no estás trabajando con Homestead, que ya te lo da todo hecho, tienes que cerciorarte de disponer el directorio

~/composer/vendor/bin en tu variable de entorno PATH, de modo que el ejecutable de Laravel se pueda localizar en cualquier lugar de tu

sistema. Será necesario cuando queramos crear la instalación del framework a partir del instalador descargado.

**2.- Instalamos una instancia del framework en nuestra carpeta de proyecto.** A través del comando "laravel new Nombre\_De\_Proyecto" creamos una instalación limpia del framework. Imagina que tu proyecto se llama "test\_desarrollo", entonces lanzarás este comando:

```
laravel new test_desarrollo
```

Eso nos creará un nuevo directorio en el sistema en el que tendremos la instalación de Laravel lista para usar. Además, todas las dependencias que usa Laravel para funcionar se instalarán en el mismo proceso, con lo que no tendrás que preocuparte por instalar por separado nada más, ni configurar ninguna librería.

Ese comando lo tienes que realizar desde la carpeta de tus proyectos, dentro de la máquina virtual de Homestead. Por ejemplo, si definiste esta configuración en el Homestead.yaml:

```
folders: - map: ~/proyectos/codigo to: /home/projects
```

```
sites: - map: mi_proyecto.dw to: /home/projects/mi_proyecto/public
```

Ese comando lo tendrás que lanzar en el directorio definido en "folders: -> to:". Dada la configuración anterior sería en /home/projects. Insistimos, dentro de la máquina virtual con Homestead.

### 1.2.4.- Alternativa tradicional de instalación de Laravel

Antes de la versión 5 Laravel se instalaba con un comando diferente, que también puedes seguir usando si es tu preferencia.

```
composer create-project laravel/laravel --prefer-dist
```

Eso se conectará con Git para traerse el código de Laravel 5 y lo copiará en tu carpeta, aunque esta opción será un poco más lenta.

### 1.2.5.- Comprobar la instalación

Finalmente queremos comprobar la instalación. Es tan sencillo como dirigirse con el navegador a la URL donde está Laravel instalado. Si obtenemos el mensaje de bienvenida, como el de esta imagen:

# Laravel 5

Very little is needed to make a happy life. - Marcus Antoninus

Si no has tenido suerte a la primera y la instalación no está funcionando no te preocupes puesto que en tu caso pueden quedar algunas cosas por hacer. De hecho lo más seguro es que sea así y que tengas que crear al menos tu archivo de entorno (.env) y la llave de la aplicación. Todo esto está detallado en el siguiente artículo en el que [explicamos posibles tareas a realizar para resolver problemas comunes](#).

Más información y actualizada en la [página de la instalación, dentro de la documentación oficial](#).

Os dejamos con este vídeo, en el que se puede ver cómo configurar Homestead y realizar la instalación de Laravel 5.

Puedes ver el vídeo en nuestro canal de Youtube. <https://www.youtube.com/user/desarrollowebcom>

Artículo por Miguel Angel Alvarez

## 1.3.- Videotutorial: Instalar Homestead y Laravel 5 en Windows

*Guía paso a paso en vídeo sobre la instalación de Laravel 5 en Windows, usando una máquina virtual Homestead, tal como se recomienda para entorno de desarrollo.*

En estos vídeos vamos a enseñar cómo instalar Homestead y cómo instalar Laravel en la máquina virtual de desarrollo (Homestead). Estos vídeos son complementarios a otros que ya hemos publicado en DesarrolloWeb.com, con la diferencia que en este caso vamos a usar el sistema operativo Windows para realizar las tareas.

¿Por qué Windows? porque indudablemente la mayoría de los desarrolladores trabajan con Windows y porque ya habíamos publicado artículos y vídeos en los que se mostraba el proceso en el sistema operativo Mac OS X. En OS X y en Linux la instalación es calcada, pero en Windows algunos lectores nos habían pedido instrucciones más precisas.

El procedimiento para instalar Laravel 5, según la recomendación oficial para entornos de desarrollo, está compuesto de dos pasos.

- Instalar Homestead (una máquina virtual de desarrollo que tiene todos los requisitos para que Laravel funcione perfectamente)
- Instalar Laravel 5, para poder comenzar el desarrollo con este framework PHP.



Estos dos pasos ya los hemos relatado en texto, con instrucciones detalladas, en artículos anteriores del [Manual de Laravel 5](#). Por ese motivo no vamos a repetir las explicaciones y vamos directamente a mostrar los vídeos.

### 1.3.1.- Instalar Homestead en Windows

En este vídeo realizamos el primero de los pasos, la instalación de Homestead, la plataforma de desarrollo basada en una virtualización de Linux con la "distro" Ubuntu. Osea, estamos diciendo que instalaremos Laravel sobre Windows, pero verdaderamente lo que vamos a crear es una máquina virtual en nuestro ordenador que tendrá el sistema operativo Linux. Homestead es el nombre que recibe el proyecto de esa máquina virtual configurada para instalar Laravel 5 para un entorno de desarrollo.

Los motivos de la instalación de Laravel sobre una máquina virtual Linux y de la existencia de Homestead en general está relatados en el artículo [Instalar Homestead](#).

Puedes ver el vídeo en nuestro canal de Youtube. <https://www.youtube.com/user/desarrollowebcom>

### 1.3.2.- Instalar Laravel 5 sobre Windows

Ahora vamos a instalar Laravel 5 en Homestead sobre una máquina anfitrión Windows. El trabajo lo hacemos sobre Homestead, por lo que este procedimiento verdaderamente sería exactamente igual en Windows, Linux o Mac, porque realmente estamos haciendo todo el proceso en la máquina virtual. Difieren pocas cosas, como la consola de comandos (terminal) que puedas usar.

El procedimiento ya se explicó en el artículo [Instalar Laravel 5](#), aunque ahora vamos a mostrar cómo se realiza todo esto de manera particular para los usuarios de Windows.

En una máquina Homestead puedes tener varios sitios web funcionando con Laravel, por lo que lo que vamos a realizar nosotros ahora es la instalación de Laravel 5 para un proyecto en particular. Luego podríamos repetir un proceso de manera similar para administrar varios proyectos como se explica en el artículo [Mantener varios proyectos con Homestead](#).

En el vídeo que puedes ver a continuación realizamos además un pequeño "Hola mundo" para saber si la instalación de Laravel fue realizada con éxito.

Puedes ver el vídeo en nuestro canal de Youtube. <https://www.youtube.com/user/desarrollowebcom>

Esperamos que estas indicaciones te hayan servido de utilidad y complementen la información presentada anteriormente sobre Laravel 5. Si tienes algún problema con la instalación, y para continuar con los siguientes pasos, te recomendamos también echar un vistazo al artículo sobre [Tareas y problemas comunes al instalar Laravel 5](#).

Acerca de problemas comunes de instalar Laravel, para la realización de estos vídeos me vi con una dificultad y es que, una vez instalado Homestead, no se iniciaba la máquina virtual y por tanto no se podía arrancar el servidor con "vagrant up". Solo tuve que actualizar la versión de VirtualBox instalada en mi ordenador, que estaba un poco viejita.

Artículo por Miguel Angel Alvarez

## 1.4.- Tareas adicionales en la instalación de Laravel 5 y problemas comunes

*Para completar la instalación tienes que realizar unas tareas adicionales, en muchos casos, así como resolver problemas comunes.*

En el artículo anterior del [Manual de Laravel](#) ya explicamos cómo realizar la [instalación de básica del framework PHP Laravel 5](#). Como has podido comprobar, instalar Laravel es muy fácil, pero en la mayoría de los casos habrá que completar algunas tareas básicas para dejarlo funcionando correctamente en nuestro sistema. En este artículo pretendemos ayudar explicando cuáles son esas posibles tareas, a la par que analizamos los problemas que te puedes encontrar durante la instalación.

Obviamente sería imposible recabar todos los tipos de problemas con los que te puedes encontrar cuando estés instalando Laravel 5, porque dependen de la configuración concreta del sistema donde lo vayamos a instalar. No obstante, vamos a hacer un listado de cosas que pueden fallar en este momento, en base a nuestra experiencia.

El temido enemigo en este caso es el mensaje "Whoops, looks like something went wrong.", que en la mayoría de los casos, en modo desarrollo, te debería advertir qué es lo que está funcionando mal. A partir de ese mensaje deberías encontrar la pista para saber como solucionar tu problema particular.

**Nota:** Además, recuerda que se recomienda instalar Laravel, al menos durante la [etapa de desarrollo, sobre Homestead](#) y ya con eso

eliminaremos la mayoría de las fuentes comunes de problemas, ya que partes de una máquina virtual que tiene todo lo necesario para que

Laravel funcione sin problemas.

Ahora van algunas de las posibles causas y soluciones de problemas que hemos detectado.



### 1.4.1.- Archivo de entorno inexistente

Hemos observado que si instalas Laravel 5 a partir del instalador (`laravel new nombre_proyecto`) no se crea el archivo de entorno. Ese es un archivo que está en la raíz del proyecto, que se llama `".env"`.

Fíjate que en la carpeta raíz debería haber un archivo llamado `.env.example`. Ese archivo es un ejemplo de configuración. La solución sería simplemente duplicar el archivo y llamarle `".env"`.

Atención aquí a la variable de entorno `APP_DEBUG`, que debería estar a `"true"` para que te de una descripción completa de los errores que se puedan producir al ejecutar Laravel. Si no está el archivo de la configuración del entorno, o dentro de él `APP_DEBUG` está a `false`, verás que el mensaje de error de Laravel no aparece con descripción detallada.

Como alternativa, via Composer también se puede acceder a la creación de este archivo de configuración del entorno. Si te fijas en el `composer.json` hay una sección de scripts que sirven para correr comandos post-instalación. Entre ellos hay uno que hace justamente la copia del `.env.example` al `.env`. El script se llama `"post-root-package-install"`. Para ejecutarlo con Composer lanzamos el comando siguiente:

```
composer run-script post-root-package-install
```

**Nota:** Todas las variables de entorno se pueden acceder via `$_ENV` que es una variable PHP superglobal, que mediante un array

asociativo te permite acceder sus elementos. Hay un helper llamado `"env"` que justamente está para facilitarte el acceso a las variables de

entorno sin usar la superglobal de PHP. Por ejemplo `env('APP_DEBUG')` te daría el valor de la variable de entorno `APP_DEBUG`.

Puedes verlo en funcionamiento en el archivo `config/app.php`.

### 1.4.2.- Permisos de las carpetas

Otra situación que puede dar lugar a errores es que no tengas permisos de escritura en las carpetas que lo necesitan. Directorios dentro de `"storage"` y `"bootstrap/cache"` deben tener permisos de escritura para el servidor web. Si has instalado Laravel en una máquina Homestead no deberías preocuparte por este detalle, pero si lo estás haciendo en una máquina distinta quizás tengas que activarlos.

Como estamos en el ordenador de desarrollo podríamos simplemente asignar `777` a los permisos y así nos aseguramos que no nos de problemas este detalle.

```
chmod -R 777 storage
```

Este asunto está documentado en la documentación oficial, aunque no sugieren poner los permisos a un valor concreto. Nosotros sugerimos solo `777` porque es tu máquina local, nunca se debería hacer eso en el servidor remoto donde va a estar la aplicación en producción.

### 1.4.3.- Llave de aplicación (Application Key)

En el caso de la instalación via el instalador de Laravel 5 también hemos observado que falta la llave de aplicación. En la documentación oficial menciona que esa llave debería haberse generado, tanto instalando con el instalador de Laravel (`laravel new`) como via Composer con la alternativa tradicional. Via composer sí se generó la llave, pero no via el instalador.

La solución es sencilla porque mediante el comando de Artisan `key:generate` se realiza todo el trabajo para ti.



**Nota:** Ojo, porque la llave se genera en el archivo de entorno .env y si ese archivo no existe quizás no funcione el comando de Artisan

key:generate. Lee el punto anterior "Archivo de entorno inexistente" para encontrar más información.

Dentro de la raíz de tu proyecto, donde se encuentra el .env lanza el comando:

```
php artisan key:generate
```

Eso te debería lanzar algo como la siguiente salida:

```
Application key [dpGvjrmnKLGszdgck1YSLrSMGeN61dy] set successfully.
```

Además el propio comando te actualiza el archivo .env, pero si no es así podrías hacerlo a mano tú mismo, en el epígrafe APP\_KEY:

```
APP_KEY=dpGvjrmnKLGszdgck1YSLrSMGeN61dy
```

Como nos avisan en la documentación, ten cuidado con este detalle porque si la llave de aplicación no ha sido generada los datos de las sesiones de usuario y otras informaciones encriptadas no estarán seguras.

#### 1.4.4.- Revisar config/app.php

En este archivo encontrarás información de configuración de la aplicación. Generalmente no necesitas tocarlo si estás impaciente y quieres ver ya el framework funcionando. Sin embargo encontrarás algunas cosas útiles como la variable "timezone" y "locale" que en una aplicación en producción desearás editar.

Otros elementos que se pueden configurar en otros archivos son Chache, Database, Session, que veremos más adelante.

#### 1.4.5.- URLs amigables a usuarios / buscadores

Laravel es capaz de mostrar las URL de aplicación de una manera amistosa para el usuario, y para los buscadores que puedan recorrer la página. Todo en Laravel comienza con un index.php que está dentro de la carpeta "public" (que debería ser tu document root). Para eliminar ese index.php de las URL y que éstas queden más limpias es posible que necesites tocar alguna cosa. Aunque si estás instalando vía Homestead no deberías encontrarte con ningún problema en este sentido.

Laravel 5 viene con un archivo .htaccess que sirve para generar las URL amigables en Apache. Solo ten en cuenta que tu apache debe tener activado el mod\_rewrite para que las redirecciones de .htaccess funcionen correctamente.

Además del .htaccess que encuentras en la carpeta "public" en la documentación de Laravel, en la sección de instalación <http://laravel.com/docs/#installation> y luego en la subsección "Pretty URL" te ofrecen una versión reducida del .htaccess que podrías probar si la que provee el framework no hace correctamente su trabajo.

Para los que están en Nginx se debe agregar una directiva en la configuración del sitio:

```
location / { try_files $uri /index.php?$query_string; }
```

Esto es todo por el momento. Si queréis profundizar en el tema os recomendamos que os veáis los vídeos que encontraréis en el siguiente enlace: [Primeros pasos con Laravel 5](#)

Artículo por Miguel Angel Alvarez

## 1.5.- Mantener varios proyectos con Homestead

*Tutorial para albergar varios proyectos realizados con Laravel en una misma máquina virtual Homestead, con varias instalaciones del framework PHP.*

Estás trabajando con Laravel y te has decantado, tal como se recomienda, trabajar en el entorno Homestead. Tienes una máquina virtual con una instalación de Laravel, pero quieres manejar diferentes proyectos, de manera independiente, reutilizando la misma máquina virtual, para no tener que gastar mayor espacio en disco o tiempo en crear una máquina nueva.

Esta es una situación muy común y que tiene una sencilla solución, simplemente a través de la configuración del archivo Homestead.yaml, creando distintos host para cada proyecto, asociados a otras carpetas del servidor.

Obviamente, para llegar a este artículo tienes que saber [Qué es Homestead](#) y saber cómo [instalar Laravel](#). Lo que vamos a aprender de nuevo es algo muy sencillo, simplemente configurar varios host virtuales para cada proyecto.



### 1.5.1.- Configuración de "sites" en Homestead.yaml

Recuerda que el archivo Homestead.yaml está en la carpeta ".homestead" que está situada en la raíz de tu usuario. Tanto en Windows, Linux o Mac: "~/homestead".

**Nota:** Solo un detalle para los de Windows, la carpeta "~/homestead" físicamente estará en una ruta del disco donde tengas instalado

Windows. Si tu usuario se llama "carlos" y has instalado Windows en el disco C, tu carpeta estaría físicamente en una ruta absoluta como

esta C:\Users\carlos\homestead. Recuerda que si usas un terminal medianamente bueno como el que te viene con Git (git bash) sí que te

reconocerá las rutas comenzando con ~/ como rutas a la carpeta de tu usuario Windows.

Dentro del Homestead.yaml está el epígrafe "sites", sobre el que podemos construir cualquier número de hosts virtuales (virtualhost en la terminología de servidores, que significa que el servidor web reconocerá esa carpeta como un sitio aparte, al que se accederá por un nombre de dominio independiente).

Si queremos crear varios host independientes, uno para cada proyecto Laravel, simplemente tenemos que listar ese número de virtualhost tal como sigue:

sites:

- map: proyecto1.local.com  
to: /home/projects/proyecto1/public
- map: proyecto2.local.com  
to: /home/projects/proyecto2/public
- map: proyecto3.local.com  
to: /home/projects/proyecto3/public

Podemos tener tantos virtualhost como queramos. Luego se trata de crear las carpetas mapeadas a cada uno de esos host virtuales en la máquina Homestead, algo que seguramente harás solamente cuando llegue el paso de la [instalación del framework Laravel 5](#) tal como se explicó anteriormente.

### 1.5.2.- Aprovisionar los cambios en el archivo Homestead.yaml

Si simplemente cambias el archivo Homestead.yaml las nuevas configuraciones no funcionarán, ni tan siquiera si reinicias la máquina virtual. Por ello tendrás que hacer un paso adicional para aprovisionar de nuevo la máquina atendiendo a los nuevos valores de configuración.

Eso se consigue con un comando como el que sigue:

```
vagrant reload --provision
```

Ese comando para la máquina virtual y luego la reconfigura, tomando los nuevos datos indicados en el archivo de configuración. Luego la vuelve a encender ya con las nuevas configuraciones.

**Nota:** Solo para mencionarlo por si a alguien se le ocurre, no hace falta hacer el `vagrant destroy` para eliminar la máquina virtual de

Homestead y luego volverla a crear. Eso podría funcionar, pero se te eliminarán los proyectos que has creado.

### 1.5.3.- Volver a configurar el archivo hosts

También podrá resultar obvio, pero hay que comentar que, inmediatamente después de crear los nuevos virtualhost y antes de poder acceder a ellos por los nombres de dominio configurados, deberás editar tu archivo hosts para agregar la IP del servidor asociada al nuevo nombre de dominio que se está definiendo. Colocarás una línea como esta:

```
192.168.10.10 proyecto2.local.com
```

También ojo con esa línea porque tu máquina Homestead podría tener otra IP distinta si lo editaste en el archivo Homestead.yaml y lógicamente el nombre del dominio para el virtualhost también será otra, que hayas configurado en tu archivo yaml.

### 1.5.4.- Conclusión

Con eso es todo! en lugar de crear máquinas nuevas para colocar proyectos distintos de Laravel, lo que generalmente harás es crear nuevos host virtuales sobre la misma máquina, así compartes el mismo entorno de desarrollo para diferentes proyectos Laravel 5.

Es una situación bastante común, pero no implica necesariamente que lo tengas que hacer así. Para un proyecto, por cualquier motivo, podrías preferir crear una máquina virtual totalmente independiente, creando una nueva instancia de Homestead, en lugar de reutilizar la que ya tenías.

Artículo por Carlos Ruiz Ruso

## Parte 2:

# Primeros pasos con Laravel

En los siguientes artículos ponemos las manos en el código para introducirnos en Laravel. Encontrarás una vista de pájaro de lo que te ofrece el framework PHP. Además te explicamos con detalle cómo funciona el sistema de rutas de Laravel y te ayudamos a realizar tus primeras páginas basadas en este sistema.

## 2.1.- Primera prueba de Laravel con el sistema de rutas

*Este sería un típico Hola Mundo realizado en Laravel, en el que podremos construir una primera ruta dentro de nuestra aplicación.*

Imaginamos que estás ansioso por pasar de la pantalla inicial de Laravel y ver ya algún tipo de salida que se haya producido mediante un código tuyo propio. En ese caso en este artículo podremos ayudarte, porque vamos a probar el sistema de enrutamiento de solicitudes Laravel y hacer una primera página que muestre una salida.

Obviamente, las cosas que vamos a hacer en este ejemplo no son del todo correctas, pero Laravel nos las permite. Nos referimos a crear salida sin usar, o dejando de lado, lo que sería el patrón de diseño MVC. Vamos a intentar ser breves, así que nos vamos a dar algunas licencias en ese sentido.

### 2.1.1.- Identificar el archivo donde se generan las rutas de la aplicación

Comenzamos viendo dónde se generan las rutas que nuestra aplicación va a responder, a través del framework Laravel. Estamos trabajando con la versión 5.1, aunque en todo Laravel 5 debe de ser igual.

Navega con tu explorador de archivos a la carpeta app/Http y encontrarás allí un fichero llamado routes.php. Ese es el archivo que contiene todas las rutas de la aplicación.



Una vez abierto observarás que tiene una llamada a un método `Route::get()`. Ese es el método que usaremos para generar las rutas.

**Nota:** El orden en el que pongamos las rutas es importante, puesto que primero se gestionarán las que aparezcan antes en el código.

### 2.1.2.- Crear nuestra primera ruta

Guiándonos por el ejemplo de ruta que encontramos podemos crear nuestra propia ruta.

```
Route::get('/test', function(){
    echo "Esto es una simple prueba!!";
});
```

Aunque tendremos que volver en breve sobre las explicaciones del sistema de rutas, es importante señalar algunos puntos.

1. Hay que fijarse que cualquier ruta corresponde con un verbo del HTTP (get, post, etc.), que es el nombre del método que estamos invocando sobre la clase `Route`. En nuestro caso usamos el método `get`, que es el más común de las acciones del protocolo HTTP.
2. Además observa que ese método recibe dos parámetros, el primero de ellos es el patrón que debe cumplirse para que esa ruta se active. En nuestra ruta hemos colocado `"/test"` que quiere decir que desde la home de la aplicación y mediante el nombre `"test"` se activará esa ruta. En este caso el patrón es una simple cadena, pero en general podrá ser mucho más complejo, generando partes de la ruta que sean parámetros variables. Todo eso lo estudiaremos más adelante.
3. Como segundo parámetro al método `get()` para definir la ruta indicamos una función con el código a ejecutar cuando Laravel tenga que procesarla. Observarás que es una "función anónima" cuyo código es un simple `echo` para generar una salida.

**Nota:** Para quien no lo sepa, una función anónima es simplemente una función que no tiene nombre. Están disponibles en PHP a partir

de la versión 5.3. Estas funciones se usan típicamente como parámetros en funciones o como valores de retorno y quizás quien venga de

Javascript las conocerá más que de sobra, porque en ese lenguaje se usan intensivamente. Otro nombre con el que te puedes referir a

funciones anónimas es "closure".

### 2.1.3.- Acceder a nuestra ruta

Viene la parte más fácil, que es acceder con el navegador a la nueva ruta que acabamos de definir. Si tu proyecto estaba en un virtualhost, usarás el nombre de tu dominio asociado a ese virtualhost, seguido con el patrón de la ruta que acabamos de crear ("/test"), quedando algo como esto:

`http://example.com/test`

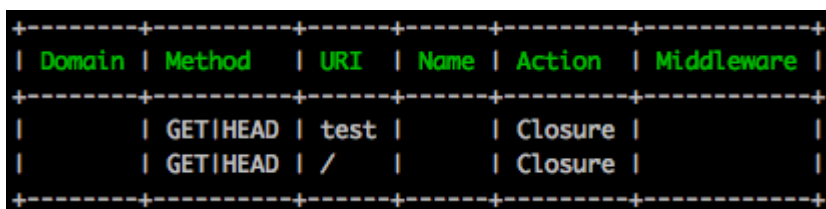
Accediendo a esa dirección deberías ver el mensaje configurado en la closure enviada al generar la ruta.

### 2.1.4.- Observar las rutas posibles dentro de una aplicación

Antes de acabar queremos comentar un comando de Artisan que sirve para mostrar todas las rutas disponibles en un proyecto o aplicación web creada con Laravel. Es un comando muy útil para saber qué tipos de rutas puedes tener y qué valores te van a permitir, parámetros, etc.

```
php artisan route:list
```

Ese comando te producirá una salida como la que puedes ver en la siguiente imagen.



Domain	Method	URI	Name	Action	Middleware
	GET HEAD	test		Closure	
	GET HEAD	/		Closure	

### 2.1.5.- Conclusión

De momento se trata de una simple ruta, pero al menos ya hemos podido tocar algo dentro del framework y cerciorarnos que todo está funcionando. En adelante podremos complicar los ejemplos todo lo que queramos y producir cualquier tipo de ruta, puesto que el sistema de routing de Laravel es muy potente.

Además, como ya hemos dejado entrever, está preparado para el trabajo con APIs REST ya que te admite directamente los distintos verbos del HTTP que se suelen usar dentro de estos sistemas. Que no te lie esto último, en realidad no es necesario construir ningún tipo de API REST para trabajar con Laravel 5, pero si lo necesitas lo tendrás bien fácil.

En el siguiente vídeo puedes ver cómo se ha creado una primera ruta como la de este artículo, simplemente de prueba, que nos sirve para saber que Laravel está funcionando correctamente.

Puedes ver el vídeo en nuestro canal de Youtube. <https://www.youtube.com/user/desarrollowebcom>

Artículo por Miguel Angel Alvarez

## 2.2.- Estructura de carpetas de Laravel 5

*Un resumen de la estructura de carpetas del framework Laravel 5, a vista de pájaro, sin entrar en demasiado detalle, pero que nos ayude a ubicar los componentes principales.*

Laravel, así como cualquier framework PHP, propone una estructura de carpetas, con la que organizar el código de los

sitios o aplicaciones web. Como ya somos desarrolladores aplicados ;) deberíamos saber que cuando más separemos el código por responsabilidades, más facilidad de mantenimiento tendrán nuestras aplicaciones. Por lo tanto, en el desarrollo del framework se cuidan mucho estos detalles, para que aquellos desarrolladores que lo utilicen sean capaces también de organizarse de una manera correcta.

En este artículo del [Manual de Laravel](#) pretendemos dar un rápido recorrido a las carpetas que vas a encontrar en la instalación de partida de Laravel 5, teniendo en cuenta que en futuras ocasiones quedará pendiente un análisis en profundidad de cada uno de estos elementos.

El objetivo de este artículo es que sepamos más o menos dónde están las cosas en el framework y que tengamos una ligerísima idea de lo que es cada cosa.



### 2.2.1.- Archivos en la carpeta raíz

Los archivos que tenemos sueltos en la carpeta raíz de Laravel son los siguientes (o al menos los más importantes que debes ir conociendo).

#### **.env**

Es la definición de las variables de entorno. Podemos tener varios entornos donde vamos a mantener la ejecución de la aplicación con varias variables que tengan valores diferentes. Temas como si estamos trabajando con el debug activado, datos de conexión con la base de datos, servidores de envío de correo, caché, etc.

**Nota:** Si .env no se ha generado en tu sistema lo puedes generar a mano mediante el archivo .env.example, renombrando ese fichero o

duplicándolo y renombrando después. Esto se explicó anteriormente en el artículo llamado [Tareas para completar la instalación y](#)

[problemas comunes](#).

#### **composer.json**

Que contiene información para Composer. Para conocer algo más de este fichero es mejor que te leas el [Tutorial de Composer](#).

Además en la raíz hay una serie de archivos que tienen que ver con Git, el readme, o del lado frontend el package.json o incluso un gulpfile.js que no vendría muy al caso comentar aquí porque no son cosas específicas de Laravel.



### 2.2.2.- Carpeta vendor

Esta carpeta contiene una cantidad de librerías externas, creadas por diversos desarrolladores que son dependencias de Laravel. La carpeta vendor no la debemos tocar para nada, porque la gestiona Composer, que es nuestro gestor de dependencias.

Si nosotros tuviésemos que usar una librería que no estuviera en la carpeta vendor la tendríamos que especificar en el archivo composer.json en el campo require. Luego hacer un "composer update" para que la nueva dependencia se instale.

### 2.2.3.- Carpeta storage

Es el sistema de almacenamiento automático del framework, donde se guardan cosas como la caché, las sesiones o las vistas, logs, etc. Esta carpeta tampoco la vamos a tocar directamente, salvo que tengamos que vaciarla para que todos esos archivos se tengan que generar de nuevo.

También podemos configurar Laravel para que use otros sistemas de almacenamiento para elementos como la caché o las sesiones.

En cuanto a las vistas cabe aclarar que no son las vistas que vamos a programar nosotros, sino las vistas una vez compiladas, algo que genera Laravel automáticamente en función de nuestras vistas que meteremos en otro lugar.

### 2.2.4.- Carpeta resources

En Laravel 5 han creado esta carpeta, englobando distintos tipos de recursos, que antes estaban dentro de la carpeta app. En resumen, en esta carpeta se guardan assets, archivos de idioma (lang) y vistas.

Dentro de views tienes las vistas que crearás tú para el desarrollo de tu aplicación. En la instalación básica encontrarás una serie de subcarpetas con diversos tipos de vistas que durante el desarrollo podrías crear, vistas de emails, errores, de autenticación. Nosotros podremos crear nuevas subcarpetas para organizar nuestras vistas. A propósito, en Laravel se usa el motor de plantillas Blade.

### 2.2.5.- Carpeta Public

Es el denominado "document root" del servidor web. Es el único subdirectorio que estará accesible desde fuera mediante el servidor web. Dentro encontrarás ya varios archivos:

#### **.htaccess**

En el caso de Apache, este es el archivo que genera las URL amigables a buscadores. <7p>

#### **favicon.ico**

Es el icono de nuestra aplicación, que usará el navegador para el título de la página o al agregar la página a favoritos.

#### **index.php**

Este es un archivo muy importante, que hace de embudo por el cual pasan todas las solicitudes a archivos dentro del dominio donde se está usando Laravel. Estaría bien que abrieras ese index.php para observar lo que tiene dentro. Para el que conozca el patrón "controlador frontal" o "front controller" cabe decir que este index.php forma parte de él.

#### **robots.txt**

Que es algo que indica las cosas que puede y no puede hacer a la araña de Google y la de otros motores de búsqueda.

En la carpeta public podrás crear todas las subcarpetas que necesites en tu sitio web para contener archivos con código Javascript, CSS, imágenes, etc.



### 2.2.6.- Carpeta database

Contiene las alimentaciones y migraciones de la base de datos que veremos más adelante.

### 2.2.7.- Carpeta bootstrap

Permite el sistema de arranque de Laravel, es otra carpeta que en principio no necesitamos tocar.

### 2.2.8.- Carpeta config

Esta carpeta contiene toda una serie de archivos de configuración. La configuración de los componentes del framework se hace por separado, por lo que encontraremos muchos archivos PHP con configuraciones específicas de varios elementos que seguramente reconoceremos fácilmente.

La configuración principal está en app.php y luego hay archivos aparte para configurar la base de datos, las sesiones, vistas, caché, mail, etc.

### 2.2.9.- Carpeta app

Es la última que nos queda y es la más importante. Tiene a su vez muchas carpetas adicionales para diversas cosas. Encuentras carpetas para comandos, para comandos de consola, control de eventos, control de excepciones, proveedores y servicios, etc.

Es relevante comentar que en esta carpeta no existe un subdirectorio específico para los modelos, sino que se colocan directamente colgando de app, como archivos sueltos.

La carpeta Http que cuelga de App contiene a su vez diversas carpetas importantes, como es el caso de aquella donde guardamos los controladores, middleware, así como el archivo de rutas de la aplicación routes.php que vimos anteriormente, entre otras cosas.

### 2.2.10.- Conclusión

Hemos podido conocer de una manera breve los componentes principales del framework Laravel. Como todo en la vida, visto así rápidamente, nos puede parecer que son muchas cosas y difíciles de entender y usar, pero no debemos preocuparnos porque los iremos conociendo poco a poco. Si tienes una base razonable de conocimientos de PHP y patrones de diseño orientado a objetos te resultará fácil usarlos leyendo los siguientes artículos del [Manual de Laravel](#).

Artículo por Miguel Angel Alvarez

## 2.3.- Verbos en las rutas de Laravel

*Explicamos cómo el HTTP Routing System de Laravel 5 permite la configuración de diversos verbos con los que especificar qué tipo de operación se desea realizar.*

El sistema de rutas de Laravel es bastante sencillo de utilizar, por lo que en pocos pasos podremos crear todo tipo de rutas bastante complejas. No obstante, al ser tan potente, si quieres analizarlo en profundidad gastarás tiempo. En este artículo pretendemos hacer una primera aproximación al sistema de enrutado de solicitudes HTTP, analizando lo que son los verbos de las solicitudes. En futuras entregas analizaremos otros elementos fundamentales.

Para comenzar no está de más una primera pasada por la documentación oficial, que de un vistazo rápido te dará una idea sobre la multitud de cosas que podrás realizar y configurar. Nosotros estamos utilizando como referencia para este artículo la [documentación del sistema de rutas de Laravel 5.1](#).

Primero queremos recordar que en el [Manual de Laravel 5](#) ya explicamos cómo realizar una primera ruta simple y vimos cosas como la configuración de las funciones anónimas o closures para especificar el tipo de tratamiento que se debe realizar para cada ruta. Esto lo puedes encontrar en el artículo [Primera prueba de Laravel](#) con el sistema de rutas y para no repetimos hay cosas que no volveremos a explicar.



### 2.3.1.- Verbos HTTP

Ya los mencionamos, pero cabe incidir de nuevo sobre los verbos HTTP porque forman parte de las rutas Laravel. Los verbos del HTTP son algo relativo al protocolo de comunicación HTTP, usado en la web, por lo tanto no tienen que ver con Laravel específicamente.

Sirven para decir el tipo de acción que quieres realizar con un recurso (la URL), siendo posible especificar en el protocolo (la formalidad de la conexión por HTTP entre distintas máquinas) el verbo o acción que deseamos realizar.

Son 8 verbos en el protocolo: Head, Get, Post, Put, Delete, Trace, Options, Connect. Y si ya eres desarrollador web y no te suena haberlos usado nunca te diré que realmente no es así. Get es la acción que se usa en el protocolo habitualmente, cuando se consulta un recurso. Sirve para recuperar información. Post por su parte es la acción que se realiza cuando se mandan datos, de un formulario generalmente. Los otros verbos no se usan de una manera muy habitual, pero sí se han dado utilidad en el desarrollo de lo que se conoce como [API REST](#).

Laravel ya viene preparado para implementar APIs REST, así que usa los verbos del protocolo para generar sus rutas de aplicación. De momento nos debe quedar claro que el acceso a un URI como "/test" puede tener diversos verbos a la hora de realizarse por parte de un sistema. Para Laravel el acceso a "/test" (o cualquier otro URI) usando la acción Get no es el mismo que el acceso a "/test" usando la acción Post.

En resumen, el significado de los verbos es:

- **Get:** recuperar información, podemos enviar datos para indicar qué se desea recuperar, pero mediante get en principio no se debería generar nada, ningún tipo de recurso en el servidor o aplicación, porque los datos se verán en la URL y puede ser inseguro.
- **Post:** enviar datos que se indicarán en la propia. Esos datos no se verán en la solicitud, puesto que viajan con la información del protocolo.
- **Put:** esto sirve para enviar un recurso, subir archivos al servidor, por ejemplo. No está activo en muchas configuraciones de servidores web. Con put se supone que los datos que estamos enviando son para que se cree algún tipo de recurso en el servidor.

- **Delete:** borrado de algo.
- **Trace, patch, link, unlink, options y connect...** no están entre los verbos comunes de Laravel, por lo que [te referimos a la Wikipedia para más información.](#)

### 2.3.2.- Registrando rutas con sus verbos

Las rutas se registran con los verbos, usando el método correspondiente de la fachada Route. Ya se ha mencionado pero lo repetimos: **Todo el listado de rutas que queramos producir se debe escribir en el fichero `app/Http/routes.php`**

**Nota:** Es la primera vez en este manual que usamos la terminología "fachada" que es un patrón de diseño de software orientado a

objetos usada de manera intensiva en Laravel. Podrás encontrar referencias a este patrón a partir de la palabra "facade", en inglés. De

momento vamos a obviar esa palabra para tratarla en detalle más adelante.

La clase Route (la fachada del sistema de rutas) tiene varios métodos estáticos con los verbos sobre los que queremos dar de alta una ruta de la aplicación. Anteriormente comentamos que estos métodos reciben dos parámetros, uno el patrón o URI que queremos registrar y otro la función con el código a ejecutar.

```
Route::get('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo
    return 'get';
});

Route::post('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo POST
    return 'post';
});

Route::put('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo PUT
    return 'put';
});

Route::delete('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo DELETE
    return 'delete';
});
```

Usar varios verbos a la vez al registrar una ruta

Como puedes imaginar, existen casos en los que te puede interesar registrar una ruta que tenga validez con varios verbos, indicando una única función que los resuelva todos. Esto se puede conseguir mediante dos métodos distintos.

```
Route::match(['get', 'post', 'put'], '/testing', function () {
    echo 'Ruta testing para los verbos GET, POST, PUT';
});
```

Como puedes apreciar, el método match requiere un parámetro adicional, el primero, en el que indicamos un array con los verbos que queremos aplicar a esta ruta.

Por otra parte tenemos el método any, que es como un comodín, que sirve para cualquier tipo de verbo del HTTP.

```
Route::any("/cualquiercosa", function(){
    echo 'La ruta /cualquiercosa asociada a cualquier verbo';
});
```

```
});
```

### 2.3.3.- Cómo probar los verbos de HTTP

Desde un navegador podrás probar fácilmente cualquier ruta con el verbo GET. Será simplemente escribir la URL en la barra de direcciones del navegador. Sin embargo, para probar el método post necesitarás crearte un formulario. Y si se trata de probar PUT o DELETE la cosa es más complicada porque necesitarías algún tipo de script especial.

Pero hay una alternativa muy cómoda que te permitirá probar cualquier tipo de verbo y enviar cualquier tipo de parámetro en la solicitud HTTP. Se trata de utilizar algún cliente Rest que podemos descargar como complemento en el navegador.

Una posibilidad es Postman, un cliente REST que es muy fácil de usar. Esta extensión disponible en [Chrome](#) o también en [Firefox](#).

A través de esta herramienta eres capaz de escribir una URL, el método o verbo de la conexión y a través del envío de la solicitud (botón send) recibir y visualizar la respuesta de Laravel.

Así que con postman ya podemos crear todo tipo de rutas en el sistema de enrutamiento y comprobar qué es lo que está pasando. Como sugerencia os indico justamente eso, probar distintos verbos, crear el código para registrar las rutas y ejecutarlas a través de Postman.

**Nota:** Si quieres probar rutas diferentes de GET observarás que no se puede. Al intentar hacer post, put o delete te sale un mensaje:

```
"TokenMismatchException".
```

Esto es debido a que en Laravel está activado un sistema antispam para solicitudes diferentes de GET, potencialmente más peligrosas, por el cual se debe comprobar un token. Hablaremos de ello más adelante. De momento vamos simplemente a desactivarlo.

En el archivo `app/Http/Kernel.php` encontrarás el "global HTTP middleware stack", osea la pila de middlewares que se ejecutan de manera global en las solicitudes HTTP. Busca la línea que hace la carga de `VerifyCsrfToken::class`, y coméntala.

```
//\App\Http\Middleware\VerifyCsrfToken::class,
```

### 2.3.4.- Conclusión

Hemos hablado de verbos en la solicitud HTTP, y a la vez hemos continuado aprendiendo del sistema de routing de Laravel. Pero nos gustaría que quedase claro que desde el navegador generalmente vas a realizar siempre solicitudes GET, o POST si envías datos de formulario.

Para usar los otros verbos que puedes controlar en Laravel realmente necesitas de un cliente REST, que sea capaz de ejecutar acciones como PUT o DELETE. Tendrán sentido usar en tus rutas cuando estés construyendo un API REST con Laravel.

Artículo por Carlos Ruiz Ruso

## 2.4.- Parámetros en las rutas de Laravel 5

*Explicaciones detalladas sobre cómo trabajar con parámetros en las rutas del framework PHP Laravel, versión 5.*

En el Manual de Laravel 5 hemos comenzado a tratar el sistema de rutas en profundidad. En el artículo anterior os hablábamos sobre las rutas y sus verbos HTTP, en este momento vamos a dedicarnos a algunos temas relacionados con el paso de parámetros.

Las rutas en las aplicaciones web corresponden con patrones en los que en algunas ocasiones se encuentran textos fijos y en otras ocasiones textos que van a ser variables. A lo largo de este artículo nos vamos a referir a esos textos variables con el nombre de parámetros y se pueden producir con una sintaxis de llaves.

Veamos varias rutas de una supuesta aplicación para entender qué es esto de los parámetros.



```
example.com/colaboradores/miguel  
example.com/colaboradores/carlos  
example.com/tienda/productos/34  
example.com/agenda/julio/2015  
example.com/categoria/php  
example.com/categoria/php/2
```

Podrás apreciar esas rutas y verás que hay zonas que son variables, por ejemplo, el nombre del colaborador que se pretende ver, el identificador del producto de una tienda, el mes y el año de una agenda, la categoría que se desea ver o la página de un hipotético listado de artículos de una categoría.

### 2.4.1.- Crear rutas con parámetros

A continuación vamos a mostrar cómo se podrían crear algunas de esas rutas anteriores en el sistema de routing de Laravel.

```
Route::get('colaboradores/{nombre}', function($nombre){  
    return "Mostrando el colaborador $nombre";  
});
```

Como puedes observar, en este ejemplo el nombre del colaborador es variable, por ello se expresa como un parámetro, encerrado entre llaves. En la función closure recibimos el parámetro y podemos trabajar con él, como con cualquier parámetro de una función.

El nombre del parámetro (variable con la que recibimos ese parámetro en la función anónima o closure) es independiente, como puedes ver en el siguiente ejemplo. Podríamos tener en el patrón de la ruta definido el {id} y luego recibir ese dato en una variable \$id\_producto. Lo que importa en este caso es el orden con el que se han definido los

parámetros en el patrón del URI.

```
Route::get('tienda/productos/{id}', function($id_producto){
    return "Mostrando el producto $id_producto de la tienda";
});
```

Se pueden enviar varios parámetros si se desea. Simplemente los recogeremos en la función, de una manera similar.

```
Route::get('agenda/{mes}/{ano}', function($mes, $ano){
    return "Viendo la agenda de $mes de $ano";
});
```

Aquí lo importante no son el nombre de las variables con las que recoges los parámetros, sino el orden en el que fueron declarados en el patrón de la URI. Primero recibimos el parámetro \$mes porque en la URI figura con anterioridad. Recuerda que el patrón era algo como "agenda/julio/2015".

### 2.4.2.- Enviar los parámetros a los controladores

Aunque todavía no hemos hablado de los controladores, queremos poner un ejemplo aquí, aunque realmente no cambia mucho sobre lo que hemos visto.

**Nota:** Aunque no hayamos explicado qué son los controladores estamos seguros que muchos de los lectores están familiarizados con

esa terminología, en ellos estamos pensando cuando escribimos estas líneas. Luego hablaremos de lo que hacen los controladores con

detalle pero para aclarar conceptos el lector que lo necesite puede ir revisando el [artículo sobre Qué es MVC](#).

Al definir una ruta a un controlador tenemos que indicarlo en el método que registra la ruta. En ese método, en lugar de definir una función anónima indicaremos el nombre y método del controlador a ejecutar.

```
Route::get('tienda/productos/{id}', 'TiendaController@producto');
```

Luego en el controlador recibimos el parámetro definido en el patrón de URI de la ruta registrada.

```
public function producto($id)
{
    return "Esto muestra un producto. Recibiendo $id";
}
```

### 2.4.3.- Parámetros opcionales

Hay ocasiones en las que se especifican parámetros que tienen valores opcionales. Mira las siguientes rutas:

```
example.com/categoria/php
example.com/categoria/php/2
```

Como puedes comprobar, a veces indicamos la página de categoría que se desea mostrar y a veces no se indica nada. Esto imitaría el funcionamiento de un paginador, en la que, si no se recibe nada, se mostraría la primera página y si se indica un número de página se mostraría esa en concreto.

Los parámetros opcionales en Laravel se indican con un símbolo de interrogación. En este código en el patrón de la URI observarás que la página es opcional.

```
Route::get('categoria/{categoria}/{pagina?}', function($categoria, $pagina = 1){
    return "Viendo categoría $categoria y página $pagina";
});
```

Resultará de utilidad, al definir el closure, indicar un valor por defecto para aquellos parámetros que son opcionales. Si no lo hacemos, a la hora de usar ese parámetro dentro de la función anónima, nos mostrará un mensaje de error en caso que no lo enviemos "Missing argument 2 for...".

#### 2.4.4.- Precedencia de las rutas

Con dos rutas registradas que tienen patrones distintos, si por un casual una URI puede encajar en el formato definido por ambos patrones, el que se ejecutará será el que primero se haya escrito en el archivo routes.php.

```
Route::get('categoria/{categoria}', function($categoria){
    return "Ruta 1- Viendo categoría $categoria y no recibo página";
});

Route::get('categoria/{categoria}/{pagina?}', function($categoria, $pagina=1){
    return "Ruta 2 - Viendo categoría $categoria y página $pagina";
});
```

En esas dos rutas tenemos patrones diferentes de URI, pero si alguien escribe:

`example.com/categoria/laravel/`

Esa URL podría casar con ambos patrones de URI. En este caso, el mensaje que obtendremos será:

Ruta 1 - Viendo categoría laravel y no recibo página

En este caso no necesitaríamos haber indicado el valor por defecto en \$pagina para la segunda ruta registrada, pues nunca se invocaría a esa ruta sin enviarle algún valor de página.

#### 2.4.5.- Aceptar solamente determinados valores de parámetros

Hay una posibilidad muy útil con los parámetros de las rutas que consiste en definir expresiones regulares para especificar qué tipo de valores aceptas en los parámetros. Por ejemplo, un identificador de producto debe ser un valor numérico o un nombre de un colaborador te debe aceptar solamente caracteres alfabéticos. Si has entendido esta situación observarás que hasta el momento, tal como hemos registrado las rutas, se aceptarían todo tipo de valores en los parámetros, generando rutas que muchas veces no deberían devolver un valor de página encontrada. Por ejemplo:

`example.com/colaboradores/666`  
`example.com/tienda/productos/kkk`

Recuerda que hemos dicho que colaboradores debería ser un valor de tipo alfabético y que el identificador de producto solo puede ser numérico. Así que vamos a modificar las rutas para poder agregarle el código que nos permita no aceptar valores que no deseamos.

```
Route::get('colaboradores/{nombre}', function($nombre){
    return "Mostrando el colaborador $nombre";
})->where(array('nombre' => '[a-zA-Z]+'));
```

Como puedes apreciar, se le coloca encadena, sobre la ruta generada, un método where() que nos permite especificar en un array todas las reglas que se le deben aplicar a cada uno de los parámetros que queramos restringir.



**Nota:** Si la ruta define varios parámetros no estamos obligados a colocar expresiones regulares para todos, solo aquellos que queramos

restringir a determinados tipos de valores.

Ahora mira este código, donde tenemos dos rutas definidas:

```
Route::get('tienda/productos/{id}', function($id_producto){  
    return "Mostrando el producto $id_producto de la tienda";  
})->where(['id' => '[0-9]+']);
```

```
Route::get('tienda/productos/{id}', 'PrimerController@test');
```

En la primera ruta estamos obligando a que el parámetro id sea un número. Sin embargo, en la segunda ruta tenemos el mismo patrón sin definir el tipo de valor al que queremos restringir. En este caso, si el id fuera un valor numérico se iría por la primera ruta y si es un valor diferente se iría por la segunda. En fin, que podemos configurar las rutas para hacer muchas cosas distintas y definir innumerables comportamientos atendiendo a las necesidades de la aplicación y a la manera en la que prefiramos organizar nuestro código.

De momento con lo que hemos visto sobre rutas podemos jugar bastante, así que para avanzar en el uso de Laravel 5 vamos a cambiar de tercio en los próximos artículos para tratar otros asuntos que a buen seguro estarás impaciente por conocer.

Artículo por Carlos Ruiz Ruso



## Parte 3:

# Introducción a los componentes principales de Laravel

En los siguientes artículos vamos a abordar, uno a uno, los componentes principales que encuentras en el framework: controladores, vistas, modelos, sistemas request y response, etc. El objetivo es que conozcas las piezas fundamentales para el desarrollo de aplicaciones web con Laravel y comiences a usarlas en ejemplos más elaborados.

## 3.1.- Introducción a las vistas en Laravel 5

*Cómo se trabaja con vistas en Laravel 5, creamos las primeras vistas y las llamamos desde el sistema de enrutado.*

En este artículo del [Manual de Laravel 5](http://desarrolloweb.com/manuales/manual-laravel-5.html) vamos a comenzar a tratar el tema de las vistas. Las vistas no son más que los archivos PHP desde donde tenemos que realizar la salida de la aplicación. Es un concepto que esperamos que ya se tenga en la cabeza cuando estamos introduciéndonos en Laravel, puesto que no pertenece en si al framework PHP sino al MVC en general.

En este artículo veremos cómo crear nuestras primeras vistas en Laravel 5 y como invocarlas para crear una salida de la aplicación web totalmente personalizada.



### 3.1.1.- Qué son las vistas

Como decíamos, la mayoría seguro entenderá este concepto, no obstante, vamos a exponerlo rápidamente para aquel que no sepa de qué estamos hablando. Las vistas son una de las capas que tiene el sistema MVC, que trata de la separación del código según sus responsabilidades. En este caso, las vistas mantienen el código de lo que sería la capa de presentación.

Como capa de presentación, las vistas se encargan de realizar la salida de la aplicación que generalmente en el caso de PHP será código HTML. Por tanto, una vista será un archivo PHP que contendrá mayoritariamente código HTML, que se enviará al navegador para que éste renderice la salida para el usuario.

**Nota:** En la práctica una vista podrá tener cualquier tipo de salida, no solo HTML. Hay ocasiones que será código PHP para generar

una imagen, un archivo de texto o cualquier otra necesidad. Por ejemplo, si ante una solicitud el servidor debe enviar como respuesta

datos en notación JSON, esos datos se escribirán mediante una vista.

### 3.1.2.- Dónde almacenar las vistas en Laravel 5

Existe una carpeta en el proyecto que es donde debemos colocar las vistas en Laravel. Está en "resources/views". Si navegas a esa carpeta observarás que dentro ya hay diversos archivos, incluso directorios. Esto es porque las vistas se pueden organizar por carpetas, para mantener agrupadas las de cada una de las secciones de la aplicación. Nosotros podemos hacer lo mismo, o dejarlas sueltas en el directorio view.

Además verás que las vistas tienen una extensión ".blade.php". Esto hace referencia a que es un archivo de salida que usa el motor de plantillas "Blade", el oficial de Laravel.

Ten en cuenta que, a pesar de la extensión ".blade.php", las vistas no dejan de ser archivos PHP donde podríamos colocar HTML plano mezclado con códigos PHP. De hecho, podríamos perfectamente nombrar a nuestras vistas como ".php" y el tratamiento que haremos para invocarlas será exactamente el mismo que si tienen la extensión ".blade.php". Es más, usar Blade es opcional, por lo que para mantener la simplicidad de nuestras primeras vistas no vamos a usar el motor de plantillas.

Así pues, para construir nuestra primera vista lo tenemos fácil. Simplemente creamos un archivo llamado "algo.php" en la carpeta "resources/views/". Dentro le colocamos cualquier documento HTML, con cualquier contenido. Con eso ya tenemos la vista lista para ser utilizada.

Como decíamos, el código de ese archivo "algo.php" de momento es indiferente, pero por si alguien necesita la

aclaración sería algo como esto:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Esto es una vista de prueba</title>
</head>
<body>
    <h1>Vista "algo"</h1>
    <p>Esta es mi primera vista en Laravel</p>
</body>
</html>
```

Como has observado, la vista no contiene ningún código PHP, porque de momento no lo necesitamos para probar. Es un HTML plano, pero no obstante, sí debemos respetar la extensión ".php" o ".blade.php".

### 3.1.3.- Cómo invocar una vista en Laravel 5

Ahora toca usar la vista creada en el paso anterior. Lo vamos a hacer desde el propio sistema de rutas, por simplificarlos la vida en este primer ejemplo. De hecho es como se invoca a la vista "welcome" en la ruta predeterminada que podremos ver al instalar el framework.

**Nota:** Aunque en una arquitectura MVC muchos desarrolladores prefieren que sean los controladores quienes invoquen a las vistas,

nosotros nos vamos a saltar de momento ese paso porque realmente Laravel no lo necesita, pero sobre todo porque todavía no hemos

llegado a explicar los controladores. En lugar de cargar la vista desde el controlador la vamos a invocar directamente desde el sistema de routing.

La corrección de esta operación sería discutible. No es lo habitual pero en ocasiones puede ser adecuada, por ejemplo que sea una vista que no necesita datos externos para mostrarse a si misma. Pero esto es otra discusión diferente al objetivo de este artículo.

En el siguiente código realizamos la invocación a una vista. Debes apreciar la función view() dentro del closure, a la que enviamos el nombre de la vista a mostrar.

```
Route::get('algo', function () {
    return view('algo');
});
```

Como has podido ver, view() es una función global, un helper global en Laravel, que se encarga de cargar una vista y devolver la salida producida por ella.

El nombre de la vista que estamos cargando es "algo". Para que no nos de un error la vista deberá estar creada dentro de la carpeta "resources/views", con esta ruta completa:

resources/views/algo.php

O bien:

resources/views/algo.blade.php

**Nota:** Si en algún momento queremos preguntar si una vista existe antes de cargarla, podemos usar la función `view()` de esta manera:

```
view()->exists('calendario.mes')
```

Si no se le envían parámetros a `view()` se recibe un objeto que tiene una serie de métodos útiles para vistas. El método `exists()` devuelve

un `true` o `false`, dependiendo de si existe o no una vista indicada.

### 3.1.4.- Organizar las vistas por carpetas

Es común que queramos poner todas las vistas que tengan que ver con la misma sección del sitio en una misma carpeta, o todas las vistas de los correos electrónicos enviados por la aplicación, por ejemplo.

Las carpetas las situaremos dentro de "resources/views" y en ellas colocaremos los archivos php de las vistas, como se ha descrito antes.

Por ejemplo crea una vista llamada `index.php` y colócala en otro directorio dentro de `views`. Su ruta sería algo como esto:

```
resources/views/otro/index.php
```

Ahora invocaremos esa vista indicando la ruta donde se encuentra, desde el directorio `views`. Omitimos de nuevo la extensión, ya sea ".php" o ".blade.php".

```
Route::get('/otro', function () {  
    return view('otro/index');  
});
```

Como alternativa podemos especificar la ruta de la vista con un punto en lugar de una barra.

```
Route::get('/otro', function () {  
    return view('otro.index');  
});
```

### 3.1.5.- Pasar datos a las vistas en Laravel

Seguro que lo siguiente que te preguntabas era cómo pasar los datos a las vistas. Si tienes que pasar datos para que las vistas los representen, los enviarás a través de la función global `view()` como un array asociativo.

```
view('calendario.eventos', [  
    'mes' => $mes,  
    'ano' => $ano,  
    'eventos' => $eventos  
]);
```

Una vez dentro de la vista los recoges en el ámbito global, a partir de las llaves de los elementos del array de datos. En

el ejemplo anterior \$mes, \$ano o \$eventos.

```
<p>
    Estás viendo el mes <?= $mes; ?> y el año <?= $ano; ?>.
</p>
```

**Nota:** Hemos hablado de las plantillas Blade, pero advirtiéndolo que las vamos a ver en detalle más adelante. Sin embargo es útil que mencionemos una estructura de sintaxis de este tipo de plantillas que verás sin duda en varios ejemplos por ahí y que nosotros también pensamos usar en el futuro, que es el volcado de datos que tienes en variables en el contenido de la vista.

El mismo código que acabamos de ver en el párrafo anterior a esta nota, en el que mostramos el valor del mes y el año, podríamos haberlo escrito así con la sintaxis de Blade:

```
<p>
```

```
    Estás viendo el mes {{$mes}} y el año {{$ano}}.
```

```
</p>
```

Como se puede apreciar, simplemente sustituyes los tag de cierre y de apertura de código PHP y el "echo" por unas dobles llaves que engloban aquello que quieres volcar a la vista. Es una sintaxis que mejora la legibilidad del código.

Para usar esa sintaxis recuerda que debes tener una plantilla Blade, por lo que el archivo lo tendrás que nombrar necesariamente con extensión ".blade.php".

De momento es todo. Con esto ya conoces lo básico de las vistas en Laravel 5. No obstante quedan muchas cosas que aprender que veremos en los siguientes artículos de este manual.

Artículo por Miguel Angel Alvarez

## 3.2.- Controladores en Laravel 5

*Explicaciones y ejemplos sobre controladores en Laravel 5. Crear controladores, invocarlos desde las rutas.*

Aunque en el [Manual de Laravel 5](#) ya nos hemos referido anteriormente a los controladores solo fue muy de pasada. En este artículo comenzaremos a trabajar con ellos, conociéndolos un poco más a fondo.

Aunque suponemos que todos los lectores deben tener unas nociones básicas generales sobre el concepto de controlador, cabe aclarar que éstos son una de las piezas que, junto con los modelos y las vistas, forman parte del patrón MVC. En Laravel, como en cualquier otro de los frameworks PHP populares, son una importante parte de las aplicaciones. Su función es la de definir el código a ejecutar como comportamiento frente a una acción solicitada dentro de la aplicación.



Generalmente para poder desempeñar su labor se apoyan en los modelos y las vistas. El controlador sabe qué métodos del modelo debe invocar, ya sea para actualizar cierta información o para obtener ciertos datos, así como las vistas que deben presentar la información como respuesta al usuario, después de la realización de las acciones necesarias.

**Nota:** Para más información sobre MVC te referimos al artículo [Qué es MVC](#).

### 3.2.1.- Controladores en Laravel

**Los controladores están localizados en la carpeta `app/Http/Controllers`** y podemos organizarlos en subcarpetas si lo deseamos. Como otras clases de Laravel están dentro del sistema de autocarga de clases, por lo que estarán disponibles siempre que los necesitemos en la aplicación.

A continuación puedes encontrar el código de un controlador básico. Merece la pena pararse a analizarlo brevemente para comentar algunos detalles.

```
<?php
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class ArticulosController extends Controller
{
    public function ver($id)
    {
        return view('articulos.ver', ['id' => $id]);
    }
}
```

Las dos primeras líneas son concernientes a los [espacios de nombres](#) donde estamos trabajando. Luego encontramos la

propia clase que define el controlador, en este caso `ArticulosController`. Como cualquier clase, por convención, usamos la primera letra del nombre en mayúscula y además en Laravel tenemos por costumbre apellidar a las clases con el sufijo "Controller". También es una convención que te ayudará a ti a identificar el código de los controladores y a otras personas que puedan trabajar en el proyecto.

Dentro de la clase colocamos las acciones que deseemos. Cada acción la implementamos a partir de un método, al que podemos invocar desde el sistema de routing, como se verá a continuación.

**Nota:** Dentro del código de la acción de momento estamos colocando una llamada a una vista, que supuestamente mostraría un

artículo. Generalmente los controladores acceden primero a los modelos para recuperar la información que se les debe pasar a las vistas

para que representen. Aquí le estamos pasando como dato a la vista simplemente el id del artículo, lo que no resultaría de mucha utilidad,

pero más adelante aprenderemos a acceder a la base de datos para recuperar esa información y poderla pasar completa hacia la vista.

Otro detalle que se puede observar es que los controladores son clases, de programación orientada a objetos, y como tales podríamos incluir en ellas cualquier miembro/s que consideremos oportuno para el funcionamiento interno, como métodos privados, propiedades, etc.

### 3.2.2.- Invocar un controlador desde el sistema de rutas

A los controladores los vamos a invocar normalmente desde el sistema de rutas, indicando el nombre del controlador y la acción (método) que debe ejecutarse para procesar una solicitud. En la siguiente línea de código registramos una ruta que llamaría a la acción "ver" sobre el controlador "ArticulosController".

```
Route::get('articulos/{id}', 'ArticulosController@ver');
```

Las acciones pueden recibir parámetros, tal como se explicó cuando [aprendimos a crear nuestras rutas en Laravel](#). En esa ruta estaríamos definiendo que se debe recibir el parámetro {id}. El valor de ese parámetro será pasado al método o acción `ver()` de "ArticulosController". Obviamente, podemos pasar tantos parámetros como sea necesario y los recibiremos en la acción del controlador en el mismo orden como fueron definidos en el patrón de la URI registrada.

### 3.2.3.- Generar los controladores con automaticamente con artisan

Crear desde cero un controlador es una tarea repetitiva dentro de Laravel, por lo que existen atajos. El ya conocido comando "artisan" nos ofrece una utilidad para crear una nueva clase controlador de una manera automática. Para ejecutarlo lanzamos en la consola este comando.

```
php artisan make:controller CategoriasController
```

Dentro de la carpeta del proyecto, invocas a artisan. La operación solicitada es `make:controller` y luego le indicamos el nombre del controlador a crear, en este caso "CategoriasController".

**Nota:** Si en cualquier momento queremos ver la lista de utilidades que nos facilita artisan, escribimos este comando en la consola:

```
php artisan
```



Pero ojo, para que funcione artisan tienes que estar en la home del proyecto, donde verás que se encuentra un archivo llamado "artisan".

Ahora verás, en la carpeta de los controladores ("app/Http/Controllers"), que ha aparecido el nuevo archivo del controlador, generado automáticamente. Te darás cuenta que tiene ya una serie de acciones definidas que son las típicas cuando quieres crear un recurso "RESTful". De momento no las usaremos pero más adelante verás que es un atajo potente, ya que te dan declarados todos los métodos que necesitarías para realizar las operaciones típicas con un recurso en un API Rest.

Artículo por Carlos Ruiz Ruso

## 3.3.- HTTP Request en Laravel 5

*Laravel nos facilita todos los datos de la solicitud actual a través HTTP Request, un objeto sobre el que podremos consultar información sobre el cliente que realiza la solicitud y datos que pueda estar enviando.*

Toda aplicación web recibe solicitudes para completar todo tipo de acciones. Cada solicitud que recibe el servidor viene acompañada de una serie de datos, que se envían en el protocolo HTTP. Entre la información que recibe PHP podemos encontrar desde el user-agent del visitante o su IP, hasta datos que viajan en las cabeceras ante una operación post.

Esos datos en Laravel se acceden a través del objeto Request, que podemos recibir en el controlador mediante la inyección de dependencias, o mediante la correspondiente facade. En este artículo del [Manual de Laravel](#) te vamos a explicar las bases del trabajo con el objeto Request.



### 3.3.1.- Recibir la solicitud (request)

Comencemos observando cómo se consigue el objeto Request en un controlador, para lo que vamos a hacer mención a un patrón de diseño de programación orientada a objetos, usado en Laravel así como en otra serie de frameworks populares: [Inyección de dependencias](#). En este patrón se busca separar la responsabilidad de creación de los objetos de su uso, simplificando y abstrayéndonos de toda la complejidad que puede suponer crear todas y cada una de las dependencias que tenga un código. Cuando una clase necesite de un objeto para completar sus operaciones no lo construye él, sino que lo recibe en el constructor o en los métodos que lo necesiten.

El hecho de recibir los objetos de los que depende una clase por parámetro es lo que se conoce como inyección. La dependencia es aquello que necesita, que no es más que un objeto de una clase. Además el patrón incluye lo que se llama el contenedor de dependencias o contenedor de servicios, que es la pieza de software encargado de instanciar los objetos que se deben inyectar e inyectarlos en los métodos necesarios. Como este patrón ya lo hemos explicado



anteriormente en Desarrolloweb.com nos vamos a ahorrar volver otra vez sobre lo mismo, remitiendo a los interesados al artículo enlazado en el párrafo anterior.

La clase Request está en el namespace "Illuminate\Http\Request", por lo que un primer paso sería declarar que vamos a usarlo.

```
use Illuminate\Http\Request;
```

A continuación podemos definir, en la cabecera del método donde queramos recibir el objeto request, la correspondiente dependencia, de la clase Illuminate\Http\Request.

```
public function recibirPost(Request $request){  
    // en este punto del código $request es mi objeto HTTP Request. (inyectado)  
}
```

Como puedes ver, para que el inyector de dependencias sepa qué es lo que debe inyectar, estamos definiendo la clase del parámetro \$request que no es otra que la clase Request.

**Nota:** La posibilidad de escribir el tipo de datos de los parámetros de métodos o funciones es algo relativamente nuevo en PHP y toma

el nombre de Type Hinting, "Implicación de tipos" en español.

Al informar el tipo observarás que no hemos colocado toda la ruta en la jerarquía del namespace "Illuminate\Http\Request", el motivo es simplemente porque ya habíamos definido un alias de esa clase mediante la sentencia "use Illuminate\Http\Request".

**Nota:** Si no hiciéramos el alias del namespace con la sentencia use Illuminate\Http\Request, entonces estaríamos obligados a definir la

jerarquía de espacios de nombres donde está situada la clase Request, con un código como el que puedes ver a continuación.

```
public function recibirPost(\Illuminate\Http\Request $request){
```

```
    // Esto es equivalente y funcionará aunque no hagas el "use"
```

```
}
```

Suponemos que estas nociones de espacios de nombres las tendrás claras, si no es así te recomendamos la lectura de los artículos de

Desarrolloweb.com sobre [Namespaces en PHP](#).

En otros artículos también veremos cómo trabajar con el HTTP Request sin necesidad de la inyección de dependencias, directamente a través de la "facade" (fachada) Request.

### 3.3.2.- Usar el HTTP Request

Una vez ya disponemos del objeto request podemos consultar toda la información disponible acerca de la solicitud. Es tan sencillo como enviar mensajes, invocando sus métodos. En este artículo haremos un ejercicio básico de envío por post, pero antes vamos a probar un par de métodos sencillos.

1. Para recuperar la URI actual de una solicitud (lo que va después del dominio), sobre nuestro objeto request invocaremos el método `path()`.
2. Para recuperar la URL completa de una solicitud (toda la ruta completa, incluyendo el dominio), invocamos el método `url()`.

Así quedaría un método de un controlador que mostrase ambas informaciones de la solicitud.

```
public function mostrarUriUrl(Request $request){  
    echo $request->path();  
    echo "<br>";  
    echo $request->url();  
}
```

Ahora vamos a ver el procedimiento completo, desde la creación de la ruta hasta la codificación del controlador, para recuperar un dato que enviarían por post en una solicitud, recogiendo los valores enviados usando HTTP Request.

En primer paso, debo de registrar una ruta en mi sistema, archivo `routes.php`.

```
Route::post('recibir', 'PrimerController@recibirPost');
```

Segundo paso defino el método `recibirPost`, que debemos de crear dentro de `PrimerController` (es un controlador que creamos en un ejemplo anterior de este manual de Laravel). En ese método inyecto la dependencia y la recibo con `$request`.

```
public function recibirPost(Request $request){  
    // código de mi método  
}
```

Como tercer paso, ya en el código de implementación del método, puedo acceder a uno de los campos que me envíen por post, de manera independiente.

```
public function recibirPost(Request $request){  
    echo $request->input('id');  
}
```

En el caso anterior estamos accediendo a un campo enviado por formulario llamado "id". Pero también existen métodos para recuperar de una sola vez todos los datos enviados en la solicitud.

```
public function recibirPost(Request $request){  
    $todos_los_datos = $request->all();  
}
```

En este caso recuperas todos los datos en forma de array, por lo que si deseas ver su contenido tendrás que usar una función como `print_r()` o `var_dump()`, o quizás mejor `dd()` que es una función específica de Laravel que muestra el contenido de una variable y a continuación para la ejecución del script.

```
public function recibirPost(Request $request){  
    dd($request->all());  
}
```

**Nota:** Para poner en marcha este ejemplo de una manera cómoda tendrás que usar una extensión como postman, que te permite generar formularios al vuelo. Algo que se explicó en el artículo Verbos en las rutas de Laravel. Otra cosa que tendrás que hacer será comentar la línea donde se accede al middleware de protección csrf, también explicado en el mencionado artículo.

### 3.3.3.- Recibir parámetros de la ruta

Aunque estemos inyectando en el controlador el objeto de la solicitud, clase HTTP Request, no impide recibir otros parámetros en el método de la acción. Esto lo podemos conseguir simplemente creando los parámetros en la acción como siempre.

Nuestra ruta sería algo así:

```
Route::post('editar/{id}', 'PrimerController@editar');
```

Ahora podremos recibir tanto la request como el parámetro de la ruta {id} en el método de la acción del controlador.

```
public function editar(Request $request, $id){  
    echo "Recibo $id como parámetro de la ruta."  
    echo "Además recibimos estos datos por formulario: " . implode(' ',  
$request->all());  
}
```

### 3.3.4.- Conclusión

Con lo que hemos visto tenemos material suficiente para poder continuar viendo otras utilidades del framework Laravel. Por supuesto, más adelante volveremos sobre el tema de las solicitudes, pues es muy importante.

Artículo por Miguel Angel Alvarez

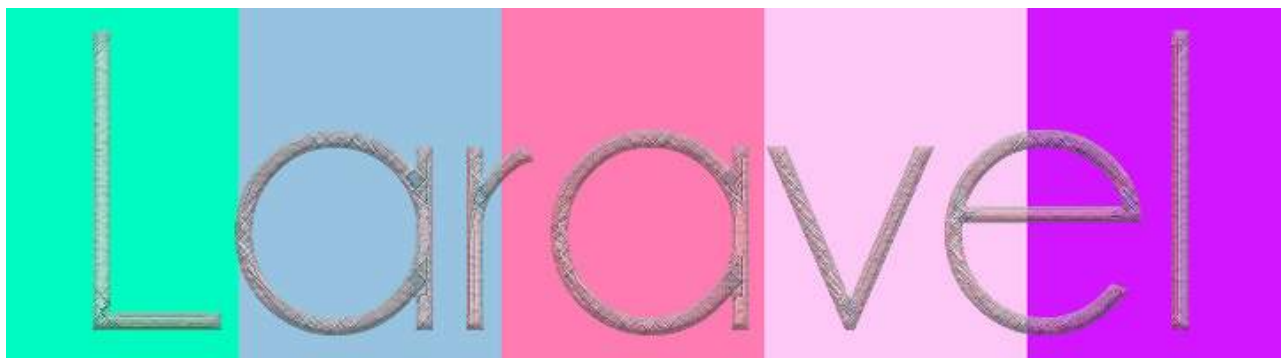
## 3.4.- Introducción a modelos en Laravel

*Introducción a los modelos, parte del patrón MVC, en el framework PHP Laravel 5.*

Siguiendo con una introducción básica a los componentes principales de Laravel, queremos hacer una primera aproximación a los modelos, de los que no habíamos hablado todavía en el [Manual de Laravel 5](#). Pero antes que nada, conviene hacer una aclaración conceptual sobre qué es un modelo:

Los modelos son uno de los componentes principales de las aplicaciones desarrolladas bajo el patrón MVC, que tienen la responsabilidad de acceder a los datos, modificarlos, etc. En el patrón además los modelos mantienen lo que se llama la lógica de negocio, que son las reglas que deben cumplirse para trabajar con los datos.

Por tanto, el tipo de acciones que le vamos a solicitar a un modelo es por ejemplo, obtener datos, insertarlos, modificarlos, etc. En las operaciones que modifiquen los datos además se tendrá que realizar cierta validación de esos datos, para asegurarnos que tienen la forma que es necesaria antes de guardarlos.



Cuando pensamos en modelos muchos hacemos una conexión directa con la base de datos: "un modelo guarda el código de acceso a la base de datos". Pero no es exactamente así, ya que un modelo trabaja con datos que pueden venir de varias fuentes. Generalmente será la base de datos, pero podría ser un API, Servicio web, sistema de archivos, etc.

### 3.4.1.- Modelos en Laravel 5

En Laravel 5 los modelos se gestionan en la carpeta "app", colocando los archivos de nuestro modelo sueltos ahí. En la instalación limpia de Laravel 5.1 tenemos un primer modelo que podemos abrir para echar un primer vistazo rápido sobre ellos. Es el archivo que está en la ruta "app/User.php".

**Nota:** Una de las modificaciones principales que aparecieron en la versión 5 de Laravel es que han quitado la carpeta de los modelos.

Esto es porque te animan a que uses la estructura de carpetas que prefieras para los modelos. En principio pueden ir todos colgando de la carpeta "app", pero también podría crearse una carpeta "models" para situarlos allí, o crear cualquier otra estructura si lo ves conveniente.

Laravel además separa código que en el patrón MVC se ubica en la responsabilidad del modelo en diversas clases dispersas por diversos directorios. Por ejemplo, las validaciones y filtrado de los datos que se reciben se pueden colocar en el middleware o en los archivos de Requests para validación que no hemos visto todavía. Así que, los que conocemos otras arquitecturas de aplicaciones basadas en MVC debemos de abrir un poco la mente cuando entramos en Laravel.

Como puedes ver, los modelos tienen la primera letra en mayúscula, por implementarse mediante clases. Los archivos donde guardamos el código de los modelos también deben tener esa primera letra en mayúscula.

En Laravel los modelos se controlan por un ORM llamado Eloquent, al menos los modelos que están implementados como datos en una base de datos, pero no es un requisito, de modo que podríamos trabajar con otros ORM o incluso bajar a un nivel más bajo y trabajar con PDO directamente, o con las extensiones de nuestra base de datos en particular.

En el caso de ser un modelo Eloquent, los modelos están directamente asociados a una entidad y a su vez a una tabla de la base de datos, por lo que un modelo que se llama User está directamente relacionado con una tabla llamada con el mismo nombre en la base de datos, pero en minúscula y acabado en plural, ej "users".

Como siempre, te recomendamos comenzar por abrir el mencionado archivo con el modelo User.php para ver cómo se

implementan en Laravel. Ese modelo está bien pero tiene un par de modificaciones un poco avanzadas que nos pueden desviar, así que preferimos explicarte los modelos con respecto a un ejemplo más vacío.

### 3.4.2.- Crear un modelo vacío en Laravel

Como en otras ocasiones, podemos ayudarnos de artisan para crear un modelo de partida. Con el comando `make:model`, seguido del nombre del nuevo modelo, creamos un modelo vacío.

```
php artisan make:model Article
```

Eso nos crea en el directorio "app" el correspondiente modelo de Eloquent, que contiene un código como el que puedes ver a continuación.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
}
```

Eso es todo lo que necesita un modelo básico en Laravel. Como te puedes imaginar, la explicación de su sencillez es que esta clase extiende la clase `Model` de Laravel.

Estos modelos ya vienen con funcionalidades para solicitar datos que estén en la base de datos. Los modelos de Eloquent estarán asociados directamente con una tabla llamada "articles", sin que tengamos que configurar nada en nuestro código, aunque más adelante aprenderemos a cambiar el nombre de la tabla asociada a un modelo, por si nos resultase necesario.

Este modelo usará el motor del ORM de Eloquent y lo puedes ver porque está haciendo uso de la clase `Model` que está en el namespace `Illuminate\Database\Eloquent`. Esa clase se le asigna un alias llamado "Model" (el mismo nombre de la clase que luego hacemos el `extends`) gracias a la sentencia:

```
use Illuminate\Database\Eloquent\Model;
```

**Nota:** Si nuestro modelo trabajase con otro ORM, o con otra base de datos que no soporte Eloquent como MondoDB, no usaríamos esa

clase `Model` para extenderlo, sino otra, y por tanto el trabajo sería diferente al que realizamos con Eloquent.

Fíjate también que el modelo se crea dentro del namespace "App", definido por la primera línea de código:

```
namespace App;
Acceder a datos del modelo
```

Desde los controladores querremos acceder a datos que mantienen los modelos: consultas, modificaciones, etc. Esas operaciones se hacen a través de la clase del modelo que acabamos de implementar.

De momento veamos cómo implementar una selección de todos los datos que tenemos en el modelo, invocando el método `all()` sobre el modelo que acabamos de crear.

```
\App\Article::all()
```

Este código estaría en un controlador, o en otra clase desde la que queramos acceder a los datos del modelo. Como puedes ver, para referirnos al modelo debemos indicar el espacio de nombres donde lo podemos encontrar, que en nuestro caso era "App".

Esa línea de código, como decíamos, nos devuelve una colección con los datos encontrados. Aunque de momento todavía no nos va a funcionar, porque la tabla "articles" no está creada en nuestro sistema gestor. En cambio obtendremos un error como este: "[...] Base table or view not found: 1146 Table 'proyecto.articles' [...]".

En futuros artículos veremos cómo crear nuestras tablas, con el sistema de migraciones y podremos comenzar a usar más a fondo los modelos. Pero como seguro estamos impacientes por comprobar si esa instrucción verdaderamente funciona, vamos a adelantar alguna cosa.

### 3.4.3.- Crear una tabla manualmente de MySQL

Podemos crear manualmente la tabla que estamos necesitando en la base de datos. Como decimos, no sería el modo correcto de proceder pero de momento con lo que sabemos vamos a conformarnos. Usaremos nuestro cliente MySQL de preferencia, como MySQL Workbench, Sequel Pro o incluso podríamos instalar PhpMyAdmin. Nosotros no vamos a usar ninguna de esas posibilidades, sino que vamos a [conectar MySQL por línea de comandos](#), que así no hay manera de fallar.

A continuación realizaremos una pequeña recetilla para crear esa tabla, que nos servirá para explicar el proceso.

1. Primero arrancamos la máquina virtual, si no estaba ya: (desde el directorio de homestead en tu disco local)

```
vagrant up
```

2. Conectas por SSH con la máquina Homestead.

```
vagrant ssh
```

3. Conectas con MySQL por línea de comandos. El host es "localhost", el usuario es "homestead" y la clave es "secret".

```
mysql -h localhost -u homestead -p
```

**Nota:** Puedes mirar el usuario y contraseña de la base de datos en el archivo .env que está en la raíz del proyecto. Otras

configuraciones de bases de datos, como el sistema gestor de base de datos usado se indican en archivo config/database.php.

Por defecto Laravel en Homestead viene configurado para usar MySQL, pero hay otros motores de base de datos que se

encuentran instalados en la máquina virtual.

4. Ya dentro del cliente MySQL por línea de comandos lanzas el comando para usar la base de datos que tengas creada.

```
use homestead;
```

5. Ahora creas la tabla y los datos de prueba.

```
CREATE TABLE `articles` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(200) COLLATE utf8_unicode_ci DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;  
  
INSERT INTO `articles` (`id`, `name`)  
VALUES  
  (1, 'Probando'),  
  (2, 'Algo'),  
  (3, 'Lindo');
```

Teóricamente ahora ya podrás acceder a la página de antes, donde habías puesto en el controlador la instrucción para mostrar todos los artículos. Solo recuerda que para mostrar la salida por la página y así poder leerla debes hacer un `print_r()` o `var_dump()` porque es una colección. También puedes usar la función `dd()` que te ofrece Laravel.

```
dd(\App\Article::all());
```

**Nota:** Realmente no necesitas ni usar un controlador, podrías hacerlo directamente con una closure dentro del sistema de rutas.

```
Route::get('articulos', function(){
```

```
    dd(\App\Article::all());
```

```
});
```

### 3.4.4.- Conclusión

Insistimos en que más adelante vamos a conocer mecanismos por los que se crean las tablas o se insertan datos de prueba directamente desde Laravel, cuando hablemos de "migrations y seeders". Aunque para trabajar en Laravel podríamos tener el schema de la base de datos hecho a mano directamente con SQL en el gestor de base de datos que estemos usando, no es la manera más habitual de proceder.

Además, hay otros métodos de acceder al sistema gestor de base de datos, como ya hemos advertido, con programas profesionales como MySQL Workbench que dan muchas mejores prestaciones y aumentan la productividad, en comparación con trabajar directamente por el terminal.

De momento creemos que es suficiente para cumplir con lo que sabemos nuestro objetivo de poner en marcha esa

llamada al modelo y recuperar información que hay en MySQL.

Artículo por Carlos Ruiz Ruso

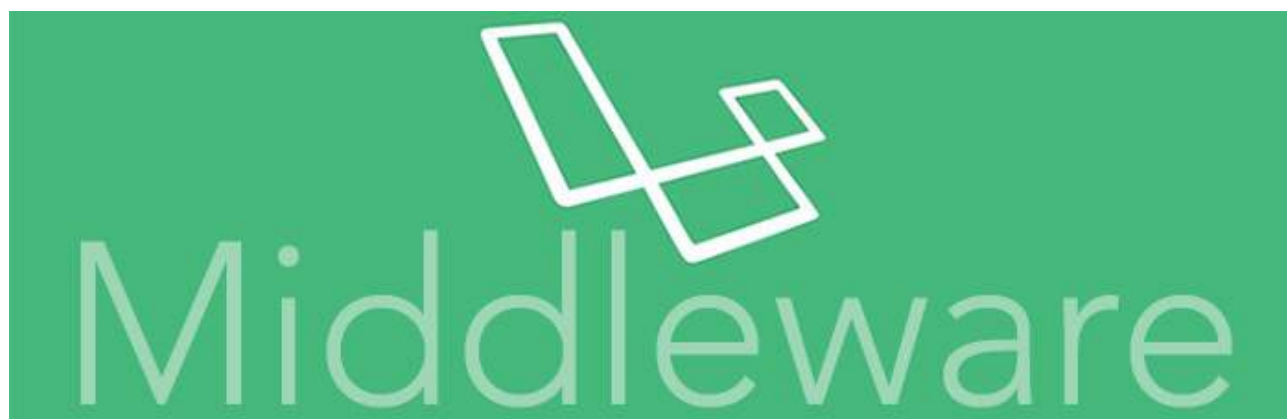
## 3.5.- Laravel middleware

*Qué son los Http Middleware, una de las piezas principales del framework PHP Laravel. Cómo trabajar con Middlewares en Laravel, creando uno nuevo.*

En el [Manual de Laravel](#) hemos podido conocer varias de las principales capas de aplicación. Hemos preferido comenzar por describir las partes más sencillas y de las cuales estamos seguros muchos tenían nociones, como son las vistas o controladores. Del MVC nos faltan por ver los modelos, pero antes de ponernos con ellos nos vamos a detener en los middleware.

Hasta ahora hemos visto que desde el sistema de routing podemos invocar a los controladores, o incluso también a través de un "closure" a las vistas. Sin embargo no habíamos mencionado que, antes llegar el flujo de ejecución a éstos hay una capa intermedia que se ejecuta en toda solicitud: el middleware.

**Los middleware en términos generales son partes del software que actúan de mediadores** entre distintos actores, permanecen en medio para facilitar la interacción o comunicación entre distintas partes de un sistema. Aquí en Laravel se quedan entre medias del sistema de routing y los controladores, permitiendo hacer cosas al pasar de unos a otros, generalmente operaciones de filtrado.



Según la definición de la documentación oficial de Laravel, "HTTP middleware provee un mecanismo adecuado para filtrar solicitudes HTTP entrantes a la aplicación [...] hay diversos middleware incluidos en el framework Laravel, como middleware para mantenimiento, autenticación, protección CSRF mediante token, etc."

Además nosotros podemos hacer nuestros propios middleware para diversas tareas, como añadir una salida en las cabeceras de toda respuesta, realizar un log de todas las solicitudes al servidor, etc. Como puedes ver, el middleware es el sitio ideal para incluir código que quieres ejecutar al principio de toda solicitud, aunque si quieres también los puedes asociar solamente a determinadas rutas.

### 3.5.1.- Archivo Kernel.php, donde se registran los middleware

Antes de ponernos a realizar nuestro propio middleware es una buena idea echar un vistazo a los que ya tenemos



funcionando de manera predeterminada. Estamos seguros que ésto ayudará a aclarar el concepto de middleware en Laravel. Para ver los middleware configurados en nuestro sistema tenemos que entrar en el archivo `app/Http/Kernel.php`.

El Kernel es el que le dice a Laravel qué middlewares tiene que cargar. Allí encontrarás una clase que tiene registrados los middleware que se van a ejecutar en el sistema, tanto de manera global (propiedad `$middleware`) como para rutas particulares (propiedad `$routeMiddleware`).

**Nota:** Quizás recuerdas que en este archivo `Kernel.php` ya habíamos entrado anteriormente, cuando descubrimos el sistema de rutas.

En el artículo [Verbos en las rutas de Laravel](#) vimos que al realizarse una ruta de tipo post se activa una comprobación de un token,

solicitado para aumentar la seguridad frente CSRF (Cross-site request forgery o falsificación de petición desde otros sitios). Esa

comprobación se realiza en el middleware "VerifyCsrfToken". En el mencionado artículo habíamos pedido comentar esa línea, para que

no se pusiera en marcha ese middleware y poder comprobar si funcionaban las rutas post.

### 3.5.2.- Crear un middleware

Para crear un middleware podríamos tomar como punto de partida uno de los que ya vienen por defecto en Laravel, pero realmente hay una manera más apropiada, usando el asistente "artisan".

Desde la línea de comandos, en la carpeta raíz del proyecto, podemos invocar artisan y solicitarle el comando "make:middleware" indicando a continuación el nombre del middleware que se desea crear.

```
php artisan make:middleware DomingoMiddleware
```

Nuestro middleware se llama `DomingoMiddleware` y se encargará de hacer cosas cuando detecte que el día actual es un domingo. Ese comando de artisan genera un middleware básico en el que solo tenemos un método, que enseguida explicamos.

La localización del archivo que se ha creado, y en general de todos los middlewares en Laravel, está en `app/Http/Middleware/`. Allí encontrarás ahora el archivo `DomingoMiddleware` que como también corresponde con el nombre de la clase que implementa el middleware, tiene la primera letra en mayúscula.

Este es el código de nuestro primer middleware:

```
<?php

namespace App\Http\Middleware;

use Closure;

class DomingoMiddleware
{
    public function handle($request, Closure $next)
    {
        if(date('w')=='0'){
            echo "Es domingo!";
        }else{
            echo "No es domingo";
        }
    }
}
```

```
        return $next($request);  
    }  
}
```

Sobre el código anterior queremos que te fijes en el método `handle()`, que es el que se ejecutará cuando se ponga en marcha este middleware. Tiene unas cuantas cosas que conviene explicar detenidamente, pero de momento solo queremos que se aprecie el `if`, donde se comprueba si el día de la semana actual es domingo, realizando dos acciones distintas si era o no era ese día de la semana.

**Nota:** Como acción de respuesta dependiendo de si es o no domingo realizamos una salida con un `echo`. Queremos remarcar que ese

`echo` no tiene mucho sentido, porque no es el momento adecuado de lanzar salida al navegador, pero ahora mismo nos sirve en este

middleware de prueba, para verlo en funcionamiento y recibir alguna salida como respuesta. El único motivo por tanto de realizar ese

`echo` es para saber si se está ejecutando. Como ya sabes, la salida la generamos desde las vistas. En el middleware acciones típicas son

redirigir al usuario a una página o realizar cualquier tarea de mantenimiento, etc.

### 3.5.3.- Registrar el middleware de manera global

Ahora vamos a registrar el middleware para ver cómo se ejecuta en el sistema. Empezaremos registrando en modo global, para que se ejecute en todas y cada una de las solicitudes que atienda nuestra aplicación.

Esto es muy sencillo y ya lo hemos dejado entrever al principio del artículo. Se hace desde el archivo `Kernel.php` que está en la ruta `app/Http/Kernel.php`.

Hay una propiedad de la clase `Kernel` donde se coloca toda la lista de middleware a ejecutar de manera global;

```
protected $middleware = [  
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,  
    // otros middleware  
    // ...  
    \App\Http\Middleware\DomingoMiddleware::class,  
];
```

Como puedes ver, es un array en el que debemos indicar el listado de los middleware a ejecutar en cada solicitud, en el orden en el que van a ser invocados. Nosotros hemos agregado el middleware creado anteriormente en el último elemento de la lista, pero si necesitase mayor prioridad sería solo ponerlo antes en el array.

Una vez registrado el middleware de manera global podrías entrar en cualquier página de la aplicación y debería verse la salida del middleware, informando si es o no domingo.

### 3.5.4.- Registrar el middleware para una ruta determinada o un controlador

Hay varias maneras de hacer este paso, desde el sistema de rutas o incluso desde los controladores, pero siempre debemos comenzar por asignar un nombre a nuestro middleware en el archivo `Kernel.php`. Allí, en la clase `Kernel`, hay una segunda propiedad llamada `$routeMiddleware`, donde tenemos un array asociativo con los middlewares que deseamos usar en las rutas.

Cada elemento del array tiene un índice y ese índice será el nombre que le demos al middleware para referirnos a él

desde el sistema de routing.

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    // otros middleware de rutas...  
    'domingo' => \App\Http\Middleware\DomingoMiddleware::class,  
];
```

Ahora podemos ejecutar el middleware a través del índice que le hemos dado en el array anterior. Lo veremos hacer mediante tres alternativas distintas:

1. Desde el sistema de routing, al definir la ruta, podemos indicar un middleware que queremos usar. Esto lo tenemos que hacer desde el registro de rutas, realizado en el archivo routes.php, mediante una sintaxis como la que puedes ver a continuación.

```
Route::get('/test', ['middleware' => 'domingo', function(){  
    return 'Probando ruta con middleware';  
}]);
```

2. También desde el sistema de routing podemos generar un grupo de rutas donde se ejecute este middleware.

```
Route::group(['middleware' => 'domingo'], function(){  
    Route::get('/probando/ruta', function(){  
        //código a ejecutar cuando se produzca esa ruta y el verbo  
        return 'get';  
    });  
  
    Route::post('/probando/ruta', function(){  
        //código a ejecutar cuando se produzca esa ruta y el verbo POST  
        return 'post';  
    });  
});
```

Así hemos indicado dos rutas donde se ejecutará ese middleware etiquetado como "domingo".

3. Desde un controlador también podemos llamar a un middleware. Lo podemos hacer desde el constructor, por lo que ese middleware afectará a todas las acciones dentro del controlador.

```
class PrimerController extends Controller  
{  
    public function __construct(){  
        $this->middleware('domingo');  
    }  
}
```

En este caso no necesitamos mencionar el middleware desde el sistema de rutas, solo lo mencionamos desde el constructor del controlador, para ejecutarse en cualquiera de las acciones declaradas en él.

### 3.5.5.- Encadenar un middleware con el siguiente

Según la filosofía de los middlewares en Laravel, como ya se comentó, pueden existir varios que se ejecuten en cadena, uno detrás de otro, para cada solicitud. Por ello tenemos que asegurarnos que este encadenamiento se pueda producir. Realmente este trabajo nos lo dan ya hecho en el middleware que se genera con artisan, por lo que no debemos preocuparnos nosotros. Sin embargo queremos que se vea dónde se consigue ese procesamiento en cadena.

Echemos un vistazo al método `handle()` del middleware generado automáticamente:

```
public function handle($request, Closure $next)
{
    return $next($request);
}
```

Como se puede ver, en este método se devuelve con `return` una llamada a `$next()` pasándole `$request` como parámetro. Esa llamada es la que permite que se ejecute el siguiente middleware en la lista de middlewares globales y además enviarle la Request completa al siguiente middleware para que siga filtrándola.

Realmente no tenemos que hacer nada manualmente, porque Laravel ya le pasa en el método `handle` el `$request` y el `$next` como parámetro. Además, Laravel sabe si ese middleware es el último de la lista, en cuyo caso no debería haber otro encadenamiento. Simplemente sería asegurarse de dejar ese `return` que aparece de manera predeterminada, para que todo siga funcionando.

### 3.5.6.- Conclusión

Los middlewares son una herramienta potente para realizar muchos tipos de acciones. Hemos visto las posibilidades más básicas que nos ofrece Laravel 5. Para los que vengan de versiones anteriores, cabe decir que vienen a sustituir el sistema de filtrado de Laravel 4 y realizando la operación habitual de comprobación de la solicitud mediante el objeto Request visto anteriormente.

Sin embargo, las utilidades del middleware se extienden para cualquier cosa que podamos necesitar. En futuros artículos nos detendremos en otras cosas y configuraciones que se pueden realizar con los middleware.

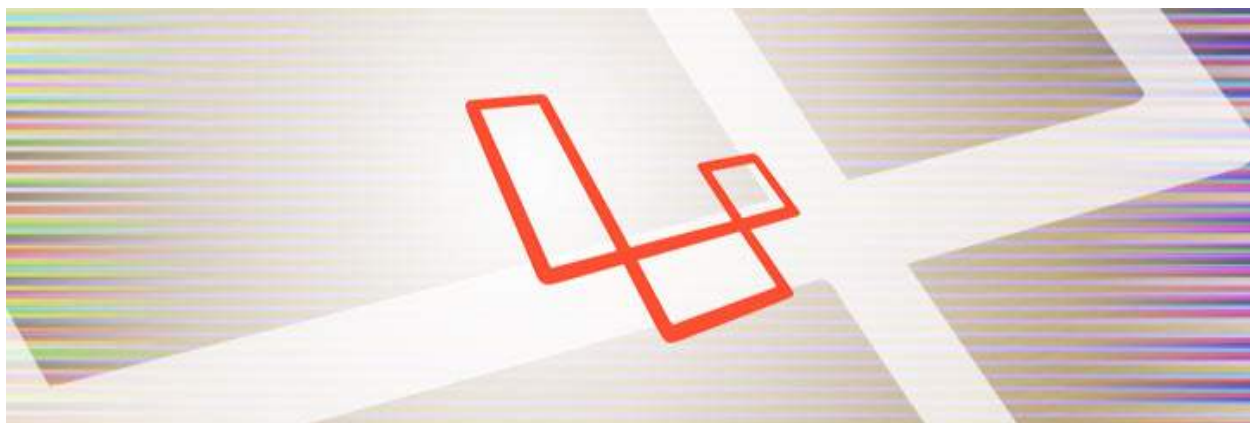
Artículo por Carlos Ruiz Ruso

## 3.6.- Responses en Laravel 5

*Qué son las responses, uno de los elementos fundamentales de Laravel 5 y algunos ejemplos de uso.*

Realmente el sistema de "Responses" es algo que abordamos en este momento de manera particular, pero que venimos usando a lo largo de diversos puntos de este manual de Laravel. Cuando devolvemos datos en forma de página web, o hacemos redirecciones por poner otro ejemplo, estamos usando el sistema response.

Las responses son las "respuestas" que nos debe devolver Laravel ante cualquier solicitud que se realice al servidor. Según la documentación oficial "Cualquier ruta [del sistema de routing] y controlador debe devolver algún tipo de respuesta al navegador del usuario". Existen diversos modos de devolver respuestas y en este artículo analizaremos algunos de ellos.



El sistema de Response depende de la clase `Illuminate\Http\Response` aunque muchas veces nos saltamos realmente el uso de esa clase y simplemente le dejamos a Laravel que la use internamente. Por ejemplo cuando escribimos salida desde el sistema de routing:

```
Route::get('ruta/de/ejemplo', function(){
    return "Respuesta desde sistema de routing.";
});
```

En este caso, como decimos, internamente Laravel recibirá esa cadena de respuesta la convertirá en un HTTP response para enviarla al cliente.

Cuando devolvemos una vista, ya sea desde un controlador o un closure en el sistema de routing también se pone en marcha el sistema de response de Laravel sin que el programador necesite usarlo directamente.

```
Route::get('cualquier/ruta', function () {
    return view('una_vista');
});
```

¿Entonces se puede abordar directamente el sistema de Response de Laravel y en ese caso, qué utilidad tiene? Efectivamente, nosotros podemos usar directamente una instancia de Response y ello nos permite personalizar todavía más elementos de la respuesta HTTP, enviando códigos de status particulares o cabeceras de HTTP.

**Nota:** La clase Response de Laravel hereda directamente de la clase `Symfony\Component\HttpFoundation\Response` que ya provee

una buena cantidad de métodos para construir respuestas HTTP. En este caso nos pueden interesar las documentaciones de [Response](#)

[de Laravel](#) y por supuesto de [Response de Symfony](#).

### 3.6.1.- Ejemplo de respuesta usando una instancia de Response

A través del helper `response()` podemos crear fácilmente una instancia del objeto Response con el que podemos hacer varios ejemplos de uso de personalización de la respuesta.

En este primer ejemplo tenemos una respuesta exactamente igual a la que conseguiríamos con un simple `return 'Hola respuesta'`.

```
Route::get('respuesta', function(){
    return response('Hola respuesta', 200);
});
```

El helper `response()`, como puedes ver, recibe dos parámetros:

1. Contenido de la respuesta
2. Código de status

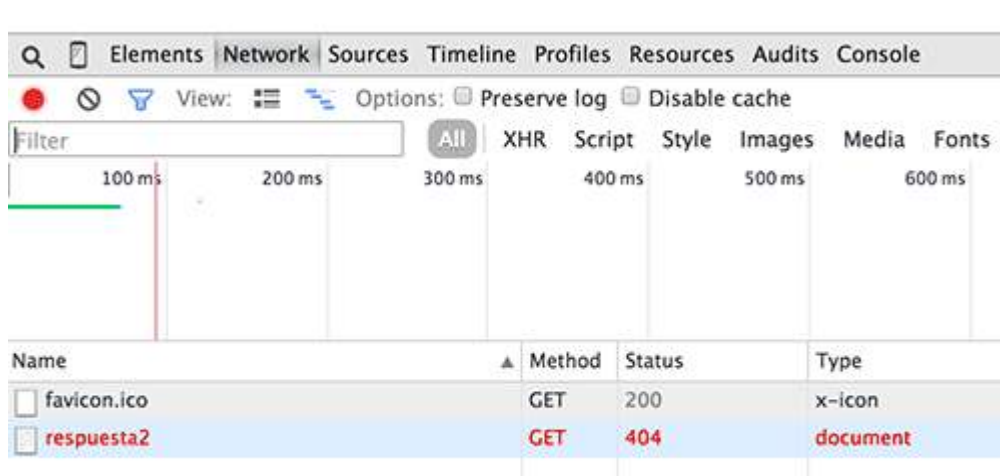
El contenido es el texto que devuelve como respuesta la solicitud HTTP. El código de status son los típicos que debes de conocer del protocolo HTTP. 200 significa "todo correcto".

Después de la invocación al helper `response()` recibimos como valor de devolución un objeto de la clase `Response`. Éste objeto es el que devuelve el closure y que Laravel toma para componer la respuesta HTTP.

En este segundo ejemplo devolvemos un error de página no encontrada, status 404, con contenido 'Esto es un error'.

```
Route::get('respuesta2', function(){
    return response('Esto es un error', 404);
});
```

**Nota:** Los códigos de status los puedes ver en las herramientas para desarrolladores de tu navegador de preferencia. En Chrome por ejemplo los podrás apreciar en la sección "Network" de las developer tools, en la columna "Status".



Si quieres conocer otros códigos de respuesta puedes consultar este artículo de la MDN: [Response codes](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

En este ejemplo realizamos algo también simple pero más elaborado, ya que enviamos una respuesta a la que le agregamos una cabecera adicional para especificar que el contenido es una hoja de cálculo, archivo CSV.

```
Route::get('respuesta3', function(){
    return response("1,2,3,4\n5,6,7,8", 200)
        ->header('Content-Type', 'text/csv');
});
```

Como decíamos, el helper `response()` devuelve un objeto de la clase `Response` y sobre él estamos encadenando la

invocación de su método `header()`, que sirve para agregar nuevas cabeceras a la respuesta HTTP. En este caso el nombre de la cabecera es 'Content-Type' y el valor que estamos aplicando es 'text/csv'.

En el ejemplo siguiente probamos otra cabecera diferente, para realizar una redirección. Tómalo simplemente como un test, puesto que existe otro método más rápido y sencillo para enviar respuestas de tipo redirect.

```
Route::get('respuesta4', function(){
    return response("", 301)
        ->header('location', 'http://desarrolloweb.com');
});
```

**Nota:** Existe un helper llamado `redirect()` que forma parte de los componentes del sistema de response de Laravel y está pensado para

hacer ese trabajo, aportando diversos tipos de redirecciones posibles. Un ejemplo podría ser este:

```
return redirect('/uri/de/redireccion');
```

En esta ocasión lo usamos para redirigir a otra URI del mismo sitio web, pero veremos otros ejemplos en futuros artículos.

Podemos encadenar varias llamadas al método `header()` si quisiéramos agregar varias cabeceras diferentes a la respuesta. como vemos en el siguiente ejemplo:

```
Route::get('respuesta5', function(){
    return response("Esta página se refrescará en 5 segundos hacia...", 200)
        ->header('Cache-Control', 'max-age=3600')
        ->header('Refresh', '5; url=http://www.desarrolloweb.com');
});
```

Hemos puesto dos cabeceras de HTTP response, la primera para definir la antigüedad máxima del elemento cacheado y la segunda para decir que en 5 segundos se refresque la página enviando al navegador a una nueva URL.

### 3.6.2.- Enviar una vista mediante una instancia Response

Habrás advertido que resulta especialmente incómodo enviar como respuesta una cadena en el primer parámetro del helper `response()`. Si la respuesta es muy larga para colocarla, así tal cual, en una cadena, o simplemente prefieres separar la salida en una vista, tal como has aprendido, tenemos una alternativa muy útil.

Se trata de invocar al helper de `response()` sin enviar ningún parámetro. La respuesta (objeto `Response`) estará limpia para configurar todos sus detalles. Uno de los métodos del objeto `Response` que te devuelve el helper `response()` es `view()`, en el que indicamos la vista que queremos procesar.

Luego podemos indicar nuevas cabeceras HTTP en la respuesta, encadenando métodos, tal como hemos visto en ejemplos anteriores en este artículo.

```
Route::get('respuesta6', function(){
    return response()
        ->view('error')
        ->header('status', 404)
        ->header('Refresh', '5; url=/');
```

```
});
```

En el código anterior tendrías un ejemplo de response en la que cargamos una vista llamada "error" y luego enviamos dos cabeceras adicionales, una para mandar un código de status (404 de error "página no encontrada") y una redirección pasados 5 segundos a la home del dominio.

### 3.6.3.- Conclusión

De momento dejamos por aquí esta introducción al sistema de Response de Laravel, hay bastante más que ver en los siguientes artículos, como responder en formato JSON, generar cookies, enviar archivos para descarga, etc. pero como nuestro objetivo de momento es la presentación de los componentes principales del framework, es más que suficiente.

Artículo por Miguel Angel Alvarez